AUG 26 1970

OCT 19 1970
NOV 12 1970

NASA TECHNICAL MEMORANDUM

NASA TM X-53962

1969

PROPERTY OF
AEDC L
F40600-71-C-0002

NASA TM X-53962

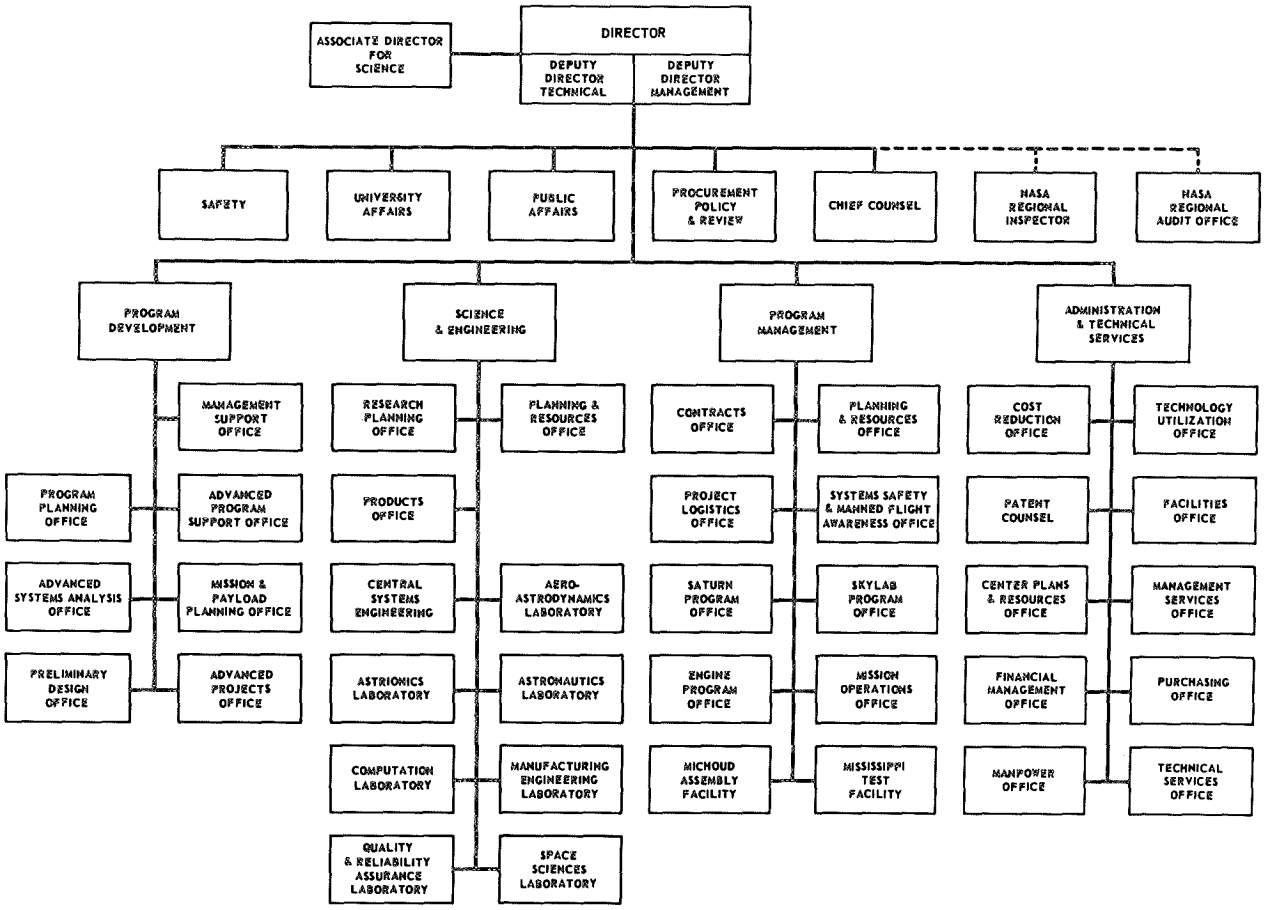
COMPUTATION RESEARCH AT MSFC

RESEARCH ACHIEVEMENTS REVIEW

VOLUME III REPORT NO. 9

SCIENCE AND ENGINEERING DIRECTORATE
GEORGE C. MARSHALL SPACE FLIGHT CENTER
MARSHALL SPACE FLIGHT CENTER, ALABAMA

GEORGE C. MARSHALL SPACE FLIGHT CENTER



RESEARCH ACHIEVEMENTS REVIEWS COVER THE FOLLOWING FIELDS OF RESEARCH

- Radiation Physics
- Thermophysics
- Chemical Propulsion
- Cryogenic Technology
- Electronics
- Control Systems
- Materials
- Manufacturing
- Ground Testing
- Quality Assurance and Checkout
- Terrestrial and Space Environment
- Aerodynamics
- Instrumentation
- Power Systems
- Guidance Concepts
- Astrodynamics
- Advanced Tracking Systems
- Communication Systems
- Structures
- Mathematics and Computation
- Advanced Propulsion
- Lunar and Meteoroid Physics

NASA TM X-53962

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

WASHINGTON, D. C.

1. Scientific research - Marshall

RESEARCH ACHIEVEMENTS REVIEW

VOLUME III REPORT NO. 9

COMPUTATION RESEARCH AT MSFC

PROPERTY OF U.S. AIR FORCE
AEDC TECHNICAL LIBRARY
ARNOLD AFB, TN 37389

SCIENCE AND ENGINEERING DIRECTORATE
GEORGE C. MARSHALL SPACE FLIGHT CENTER
MARSHALL SPACE FLIGHT CENTER, ALABAMA

1969

PREFACE

In February, 1965, Dr. Ernst Stuhlinger, now Marshall Space Flight Center's Associate Director for Science, initiated a series of Research Achievements Reviews which set forth those achievements accomplished by the laboratories of the Marshall Space Flight Center. Each review covered one or two fields of research in a form readily usable by specialists, systems engineers and program managers. The review of February 24, 1966, completed this series. Each review was documented in the "Research Achievements Review Series."

In March, 1966, a second series of Research Achievements Reviews was initiated. This second series emphasized research areas of greatest concentration of effort, of most rapid progress, or of most pertinent interest and was published as "Research Achievements Review Reports, Volume II." Volume II covered the reviews from March, 1966, through February, 1968.

This third series of Research Achievements Reviews was begun in March, 1968, and continues the concept introduced in the second series. Reviews of the third series are designated Volume III and will span the period from March, 1968, through March, 1970.

The papers in this report were presented September 25, 1969

William G. Johnson
Director
Research Planning Office

Page intentionally left blank

CONTENTS. . .

MEMORY REDUCTION THROUGH HIGHER LEVEL LANGUAGE HARDWARE

By H. Kerner and L. Gellman

	Page
ABSTRACT	1
INTRODUCTION	1
PROBLEM DEFINITION AND APPROACH	2
INSTRUCTION SET AND PREPROCESSOR	3
ORGANIZATION AND OPERATION OF THE FORTRAN LANGUAGE PROCESSOR	5
EVALUATION	8
CONCLUSIONS	12
REFERENCES	13

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Arithmetic and logic processors	2
2.	Instruction set	4
3.	FLP flow diagram	6
4.	Tradeoff equation (2), small quantity (n = 5), high reliability (r = 3)	10
5.	Tradeoff equation (2), large quantity (n = 50), high reliability (r = 3)	10
6.	Tradeoff equation (2), large quantity (n = 50), commercial reliability (r = 1)	11

FUNCTIONAL DESIGN CONSIDERATIONS FOR AN EXECUTIVE SYSTEM FOR A GENERAL PURPOSE SPACEBORNE COMPUTER

By J. R. Kennedy

	Page
ABSTRACT	15
SUMMARY	15

CONTENTS (Continued). . .

Page

INTRODUCTION. 15

FUNCTIONAL REQUIREMENTS 15

OPERATING SYSTEM FUNCTIONAL DESCRIPTION. 17

CONCLUSION 26

REFERENCES 26

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Control of system monitors and monitor intercommunications	19
2.	Simplified interpreter	21
3.	Monitor intercommunications and control task routing	22
4.	Intermonitor communications buffer format	23
5.	Launch complex computer configuration	24
6.	Event entry in mission schedule	25

PARALLEL PROCESSING METHODS AND MANNED SPACE MISSIONS

By M. E. Stegenga

Page

INTRODUCTION. 29

ARRAY PROCESSORS 29

THE HOLLAND MACHINE 31

A DISTRIBUTED PROCESSOR 33

A MULTIPLE COMPUTER SYSTEM 35

CONCLUSIONS. 35

REFERENCES 37

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Array structure of the Illiac IV system	30

Page

2. Array structure of the Holland Machine. 32

3. Distributed processor system 34

4. Multiple computer system. 35

OPTIMIZATION OF AN INSTRUCTION SET FOR A GENERAL PURPOSE SPACEBORNE COMPUTER

By J. R. Kennedy

Page

ABSTRACT 39

SUMMARY 39

INTRODUCTION 40

BASIC INSTRUCTION SET 41

IMPACT ANALYSIS 44

LIST OF TABLES

Table | Title | Page

1. Instruction Summary 45

2. Instruction Distribution 47

3. Bit Requirements and Number of Registers Provided by Several Register Allocation Schemes 50

4. Number of Bits for 16 Registers 50

5. Register Layout 51

LIST OF ILLUSTRATIONS

Figure | Title | Page

1. Computer block diagram. 40

2. Instruction word formats 43

3. Four schemes for register usage 49

Page

4. Bit cost per register for four schemes 51

5. Partially complete formats 52

EFFICIENCY AND QUEUEING TIME CALCULATIONS FOR COMPUTER BANKS

By B. G. Grunebaum

Page

ABSTRACT 57

INTRODUCTION 57

FORMULATION OF THE PROBLEM 57

RESULTS 59

NUMERICAL DETAILS 63

REFERENCES 64

BIBLIOGRAPHY 64

LIST OF ILLUSTRATIONS

Figure | Title | Page

1. Basic subsystem 58

APPLICATION OF DISCRETE DIGITAL SIMULATION LANGUAGES TO THE DESIGN OF ADAPTIVE DATA BUFFERING STRATEGIES

By L. K. Paul, Jr.

Page

ABSTRACT 65

DISCUSSION 65

CONCLUSION 68

BIBLIOGRAPHY 69

CONTENTS (Continued). . .

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Space station environment	66
2.	Abbreviated roster of GPSS II blocks	67
3.	Typical information flow problem	67
4.	Buffer problem coded in GPSS	68

AUTOMATIC MALFUNCTION ANALYSIS (AMA) FOR DISCRETE SYSTEMS

By D. T. Thomas and R. L. Jaegly

	Page
SUMMARY	71
AMA FUNCTIONS	72
THE SYSTEM MODEL	72
PREPROCESSOR PROGRAM	76
AUTOMATIC MALFUNCTION ANALYSIS	80
COMPUTER PROCESSING	82

LIST OF TABLES

Table	Title	Page
1.	Boolean Operators	73
2.	Run Analysis	83

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Hardware configuration	71
2.	Typical relay schematic	73
3.	Simple logic equation sample	74
4.	Typical series circuit	75

Page

5. Typical parallel circuit. 76

6. Simulation history printout 78

7. State list from discrete network simulation 79

8. Test procedure verification by discrete network simulation. 80

9. AMA display 81

MARSYAS — A SOFTWARE SYSTEM FOR THE DIGITAL SIMULATION OF PHYSICAL SYSTEMS

By H. Trauboth and N. Prasad

Page

SUMMARY 85

INTRODUCTION 86

SIMULATION CAPABILITY 87

ENGINEER-ORIENTED LANGUAGE 89

MATHEMATICAL FOUNDATION 90

SOFTWARE STRUCTURE 98

POTENTIALS AND IMPLEMENTATION OF MARSYAS 101

REFERENCES. 102

LIST OF TABLES

Table | Title | Page

1. Extract from List of Standard Elements. 88

LIST OF ILLUSTRATIONS

Figure | Title | Page

1. Example of a model described by a block diagram of multiple input/output blocks and a nested block 89

2. MARSYAS — Program of the example shown in Figure 1. 91

Page

3. Overview diagram of the mathematical process. 95

4. Flow of numerical solution of matrix equations. 97

5. Overview of MARSYAS systems software. 99

REAL-TIME SPACE VEHICLE AND GROUND SUPPORT SYSTEMS SOFTWARE SIMULATOR FOR LAUNCH PROGRAMS CHECKOUT

By H. Trauboth, C. O. Rigby, and P. Brown

Page

SUMMARY 105

INTRODUCTION 105

SCOPE OF SIMULATION. 106

SIMULATOR SOFTWARE 111

CONCLUSIONS 127

REFERENCES. 128

LIST OF TABLES

Table | Title | Page

1. Magnitude of Equations for Instrument Unit 110

2. Buffer Description Tables. 115

3. Pre-simulation Phase Capabilities 116

4. Format of Equation on Disc. 118

5. Equation Cross-Reference Listing 119

6. Switch and Cross-Reference Block 120

7. Initialization Phase Capabilities. 121

8. Hold Phase Commands 122

9. Hold Phase Capabilities 122

10a. Results of Successive Evaluation of Logical Equations with Two Different Initial Conditions if DO Changes State From "1" to "0" 124

CONTENTS (Continued). . .

Page

10b. Transition Table for All Possible States. 124

11. Queue Format 125

12. Real-Time Phase Capabilities 125

13. Post-Simulation Phase Capabilities 125

LIST OF ILLUSTRATIONS

Figure Title Page

1. Saturn V Launch Computer Complex configuration 107

2. Real-time simulator computer systems configuration 108

3. Example of a typical discrete/analog circuit 109

4. General flow of Simulation Processor 112

5. DDAS Simulator commutation memory tables 113

6. Master Equation Tape format 116

7. Cross-Reference File 117

8. DDAS Assignment File 117

9. Simulator flow diagram 126

STATE VARIABLE DESCRIPTOR SYSTEM (SVDS)

By N. F. Geer

Page

SUMMARY 131

INTRODUCTION 131

MATHEMATICAL MODEL 131

SYSTEM BLOCK DIAGRAM 132

PROGRAM OPERATION 134

APPLICATION 137

CONCLUSIONS 142

REFERENCES 142

LIST OF TABLES

Table	Title	Page
1.	Coding of Block Diagrams for Computer Input.	133

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Matrix block diagram	132
2.	Sample system block diagram	134
3.	Subroutine example	137
4.	Role of SVDS in dynamic system analysis.	137
5.	Numerical integration by the Runge-Kutta method	140
6.	Adjoint system	141

STORAGE SCOPE GRAPHICS AND EXPERIMENTAL APPLICATIONS

By R. Seitz

	Page
INTRODUCTION	143
TEXT AND FORM EDITING.	145
AMTRAN	148

LIST OF TABLES

Table	Title	Page
1.	Current Status of the Implementation of AMTRAN	148

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Prototype version of AMTRAN terminal.	143
2.	Space shuttle configuration	143
3.	Logic circuit.	144

CONTENTS (Concluded). . .

Page

4. Printed circuit board. 144
5. Fresnel curve 144
6. Solution of a family of differential equations 144
7. Computation Laboratory Form 1125 145
8. Schedule for branch project 145
9. Form 1122 147
10. Completed Form 1122 147
11. Editing process 147
12. Edited process displayed on 1122. 148
13. Curve with evenly spaced points 149
14. Result generated by AUTOMATH routines when applied to the curve of Figure 13. 149

MEMORY REDUCTION THROUGH HIGHER LEVEL LANGUAGE HARDWARE

By

H. Kerner and L. Gellman*

ABSTRACT

Traditional assignment of computer functions to logic and memory are evaluated based on trends in their cost and mission characteristics. The higher information content of a higher level language is exploited for the savings of memory. An instruction set was selected from the directly executable statements of FORTRAN, and the remaining statements were identified for compilation. Programs written in this language occupied 75 percent less memory than those obtained by compilation. A computer for the execution of the selected FORTRAN statements was functionally designed in order to estimate the corresponding logic hardware effort. The tradeoff between costs saved through memory reduction and expenditures for the additional logic hardware considered development, manufacturing, power, and weight costs. The proposed concept shows a cost advantage for missions with a high weight penalty and large memory requirements. Another advantage is the higher execution speed inherent in this organization.

INTRODUCTION

Innovations in hardware technology and extraordinary performance demands will prove a major factor in the organization of future spaceborne computers. Advancing semiconductor technology such as large scale integration (LSI) will make large scale computers on board spacecraft a reality. Indeed, the sophistication of future spacecraft will make such an onboard computer a necessity.

One of these future spacecraft may be an Earth-orbital space station that supports 10 or more astronauts, carries equipment and supplies for some 300 scientific experiments, and has an orbiting lifetime of several years. The computer requirements of such a mission for experiment control and data processing, attitude control, guidance and navigation, performance monitoring, maintenance support, communication, etc. can be expected to be of a magnitude equivalent to a computer with real-time characteristics supporting a large ground-based laboratory. Another spacecraft, unmanned, on a journey to Mars would require extraordinary reliability and demand computer services for data compression, guidance and navigation, and communication. Without a man on board, a large scale computer could increase the efficiency of the mission by data sampling and choice making through an adaptive mode of operation [1].

Such extraordinary reliability and durability requirements, combined with constraints on its power supply and launch costs of volume and weight, impact directly on the computer organization and may result in a substantial departure from present-day organization. In the design of spaceborne computers, these reliability and cost considerations prove especially relevant in the tradeoff between functions assigned to logic hardware and to memory. Presently, any particular balance of function allocation between logic hardware and memory is based on their respective manufacturing costs. Current trends in LSI technology promising manufacturing cost savings and weight reduction will shift that balance. These considerations demand a re-evaluation of logic-memory function assignments.

Expected LSI progress in the first half of the 1970's should reduce the weight of logic hardware by a factor of 30, while weight reduction in magnetic memories is expected to be much less drastic. This means that computers with present-day organization implemented in the technology of the mid-1970's will have the bulk of their weight concentrated in memory devices. For computers used in weight (or volume) sensitive missions, a design approach is indicated that will minimize memory at the expense of adding logic hardware. The validity of

*The authors greatly appreciate the critical review and advice of Mr. Sidney Stein and Mr. James Kennedy.

this approach is reinforced by the cost reduction expectations in logic hardware through LSI technology when compared with the limited cost reduction possibilities in memory technology.

For these reasons, this paper reinvestigates the traditional logic-memory balance that has survived three computer generations and offers a memory saving approach that exploits the higher information content of higher level languages.

PROBLEM DEFINITION AND APPROACH

Consideration of several types of computer languages provides insight into how memory savings can be realized. Computer languages can be categorized by levels, each characterized by a degree of generality:

Level	Types of Languages
L4	Problem-Oriented Languages (COGO, GPSS, -----)
L3	Procedure-Oriented Languages (FORTRAN, ALGOL, PL1, -----)
L2	Machine Peculiar Languages
L1	Microcode

A program written in a particular level language processed by the appropriate compiler produces a program in the language of the next lower level. The higher level language together with its compiler defines the detailed machine actions as completely as the compiled program does in the language of a lower level. The above mentioned points and the fact that higher level language programs occupy an order of magnitude less memory than the compiled versions demonstrate the memory saving qualities of these languages.

To exploit these language properties, a technique was required that interprets higher language statements and immediately executes an equivalent group of lower level machine actions. Since it avoids the "assembly" of the total program into the next lower level, this interpretive principle can be used to reduce memory requirements. Whether in the microprogrammed or hard-wired version, this approach is valid between any two language levels.

The goal of this study is to define a computer organization that introduces a level of control above the conventional level of arithmetic and logic processors (Fig. 1). This system features several processors on this control level capable of receiving higher level language statements, interpreting them each in their respective language, and controlling their immediate execution in lower level processors. The statement repertoire of each language and its associated control processor shall be sufficient for the description and control of complete programs or program segments.

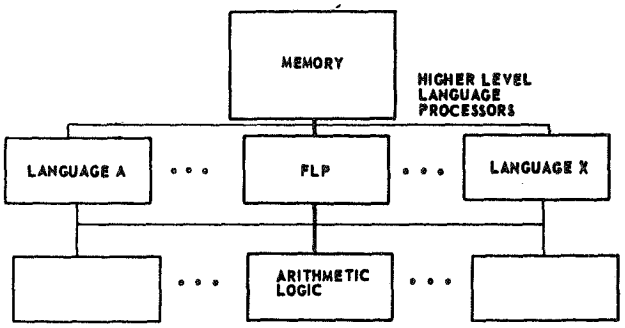


Figure 1. Arithmetic and logic processors.

The resulting computer organization will minimize computer memory by adding logic hardware. A comparison with conventional design will be made with respect to:

- manufacturing cost
- power requirement
- weight and size

} generalized cost

- speed
- reliability

For this comparison, the problem is restricted to direct code interpretation and execution between two adjacent language levels; specifically, the procedure oriented language and the machine language levels. Further restrictions limit the system to one processor on each of these levels.

The selection of the higher level language will certainly influence the result of the comparison. Languages designed primarily for numerical calculations, well matched to existing general

purpose computer organizations, offer scant opportunity for higher level execution. Therefore, languages rich in sophisticated control features are expected to yield more favorable results than languages strong in arithmetic and logic characteristics. For instance, a language designed especially for the writing of executive programs lends itself extraordinarily well to the implementation through high level control processors. Such a language would also make less use of the lower level processors.

FORTRAN IV was chosen as the higher level language for this study for the following reasons:

1. FORTRAN IV's control features are balanced with arithmetic and logic features.

2. A great variety of programs including spaceborne applications were available for evaluation of the proposed design.

3. It has wide usage and a comparison is therefore of broad interest.

The corresponding processor will be called FORTRAN Language Processor (FLP).

It is probable, however, that FORTRAN IV will not be the language best suited for spaceborne computer programming. If other, more specialized, languages prove better suited to space applications, the results of this study will represent a worst case for the proposed computer organization.

Aside from the language selected for this study, the sophistication of the higher level language processor design constitutes a second influence on the results obtained. Since the computer is destined for spacecraft application, desirable design simplification can be attained through the use of a terrestrial computer preprocessor program. Such a ground-based preprocessor would sort out the program statements, compile those statements not required for execution, and format the others for interpretative execution by the spaceborne computer. Preprocessed programs can be loaded into the computer before and during a spacecraft mission. This approach simplifies the FLP by avoiding the difficulties of a complex hardware compilation without compromising the memory saving features of the design [2-4].

Input/output (I/O) statements were eliminated from the study comparisons because of the difficulty of obtaining ground-based analogues to spaceborne applications. Terrestrial machines of the type available for the study read cards, tapes, discs, drums, and output through these media, while the spaceborne computer will collect data from sensors through various data acquisition equipment and transmit results to actuators or to data links on Earth. Future spaceborne computer I/O repertoire is expected to include the combined I/O features of ground-based process control, checkout, navigation, communications, and general-purpose computers.

INSTRUCTION SET AND PREPROCESSOR

The allocation of functions to the preprocessor and the higher level language processor was governed by the following considerations:

- The requirement for maximum memory saving with minimum additional logic hardware cost.
- That execution speed should not be less than that obtained with a conventional compiler approach.
- That speed improvement possibilities uncovered be incorporated only if they can be obtained with minor hardware penalty.

An analysis of the FORTRAN language in light of these considerations suggested the classification of FORTRAN statements into five categories; arithmetic, control, input/output, compilation aids, and comments. Compilation aids and comments are handled by the preprocessor since they do not produce executable code. From the remaining three statement types, the control and the arithmetic statements proved candidates for implementation in hardware, because every instruction of this type offers opportunity for substantial memory savings through higher level execution. For reasons previously mentioned, I/O statements were not exploited in this study, despite the fact that large memory savings and improved speeds are expected from hardware implementation of the specialized I/O requirements of spaceborne systems.

Arithmetic FORTRAN statements require special consideration. The analysis of the structure of an expression, as well as the optimization of computational sequences, does not directly contribute to memory savings. However, associated address modifications expedited by the controls of the arithmetic-logic processor and supported by the high efficiency of data transfer between the two levels of processors, are considered worthy of implementation in the higher level language processor.

The result of this analysis of FORTRAN statements resulted in the following instruction set, chosen for implementation in the higher level language processor:

1. GO TO (includes COMPUTED GO TO)
2. IF (logical and arithmetic)
3. DO
4. CALL (no argument)
5. Library Call (arguments included)
6. RETURN
7. Input/Output (READ, WRITE)
8. Parts of Arithmetic Statements:

   Address Processing

   Control of Arithmetic and Logic Processor
9. Replacement (=)

Although this list appears small, it can be verified that all basic FORTRAN IV control operations can be performed using this instruction set.

Memory reduction goals also governed the instruction word format selected. A variable subfield format was used to conserve memory despite the additional decoding required. Whenever possible, two instructions were packed into one 36-bit word. More than one half-word is required for some arithmetic and I/O statements, the library call, COMPUTED GO TO, and arithmetic IF statements.

The only common feature of all instruction words is the 3-bit operation code (Fig. 2). The right portion of the half-word contains a 13- to

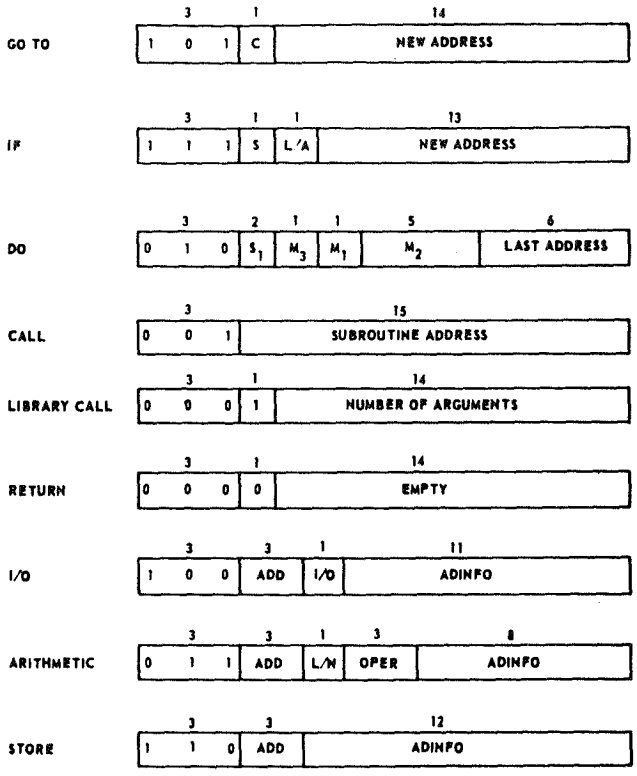


Figure 2. Instruction set.

15-bit address, or instruction modifiers supporting the operation code. The ADD field in the I/O, ARITHMETIC, and STORE instruction specifies the method of addressing data as follows:

000 Integer data accompanying the instruction

001 The actual data address in memory

010 An FLP register address

011 The previous address plus 1

100-111 Dimensioned data

Another feature employed for code compacting was the construction of standard instruction modes for the commonly expected simple cases and the indication of extraordinary cases in short fields (labeled S and S1) that control the decoding of words following in sequence.

The two approaches to memory compression discussed above, i.e., the use of a procedural language for the total program and compact coding within each specific instruction word, both require a substantial increase in logic hardware.

The search for a device to keep the hardware within bounds leads to the idea of the preprocessor executed on a ground computer, which would relieve the higher level language processor from all functions not absolutely necessary for the primary goal of code compression. This preprocessor was assigned functions of an assembler and of a compiler. The typical assembler tasks are:

1. Translation of the mnemonic code into binary code in a nearly one-to-one correspondence with each FORTRAN statement.

2. Assignment of relative addresses for program variables, instructions, and statement labels.

3. Formatting and packing of the code into the instruction word.

Preprocessor compiler tasks are:

1. Separation of FORTRAN statements directly executable by the FLP from statements requiring further analysis such as arithmetic statements.

2. Generation of additional instructions for initialization of registers before the beginning of computational sequences and for transitions from one sequence to another one.

3. Elimination of operations involving mixed modes of data types.

4. Data placement according to DATA statements.

5. Optimization of arithmetic operations and control paths.

6. Optimization of data addresses relative to the order of program execution.

ORGANIZATION AND OPERATION OF THE FORTRAN LANGUAGE PROCESSOR

A quantitative evaluation of the proposed approach toward memory reduction required the design of a language processor capable of interpreting the selected higher level instruction set. A functional design of the higher level language processor proved sufficient to estimate its hardware complexity considering the heavy influence of other factors on the tradeoff evaluation. The products of this functional design were a description of the decoding scheme for the variable format instructions, a chart depicting the flow of data between registers, and a definition of the pertinent program execution events and the corresponding control mechanism. A simplified flow diagram of the FLP illustrating the following discussion is shown in Figure 3.

Instruction execution begins as soon as one memory word (containing two instruction words) has been obtained from memory and loaded in a 32-bit instruction register with a capacity of two memory words (four instruction words). A fetch overlap feature loads the next two instruction words into this register while the first two instruction words are decoded and executed. A memory address register specifies the next (double) instruction location in memory in the usual way, while a pointer locates the instruction to be executed within the instruction register. The memory address register and the pointer combined act as an instruction location counter. The multiple instruction register allows multiple word instructions to be executed without further access to memory. For example, with the four instruction word register, DO loops of four instructions or less can be executed directly without accessing instruction memory until the loop is completed.

A special situation occurs during a fetch operation of an instruction word supported by more than four trailer instruction words. After decoding the first words, the resultant state is stored in a special register. The execution of this state is deferred until loading and decoding of the remaining instruction words have been completed.

The decoding of the variable field instructions requires a more complex decoding scheme than usually employed. The main decoder analyzes the instruction code and routes the subfields to corresponding subsections for further decoding.

A special case of a multiple word instruction is an arithmetic statement that is represented (after preprocessing) by a string of arithmetic instructions. When encountering the first arithmetic instruction of the string, the decoder initializes processing of the string by setting control flip-flops which indicate that string processing is in progress and which discriminate between logical and numerical arithmetic and between real and integer operands.

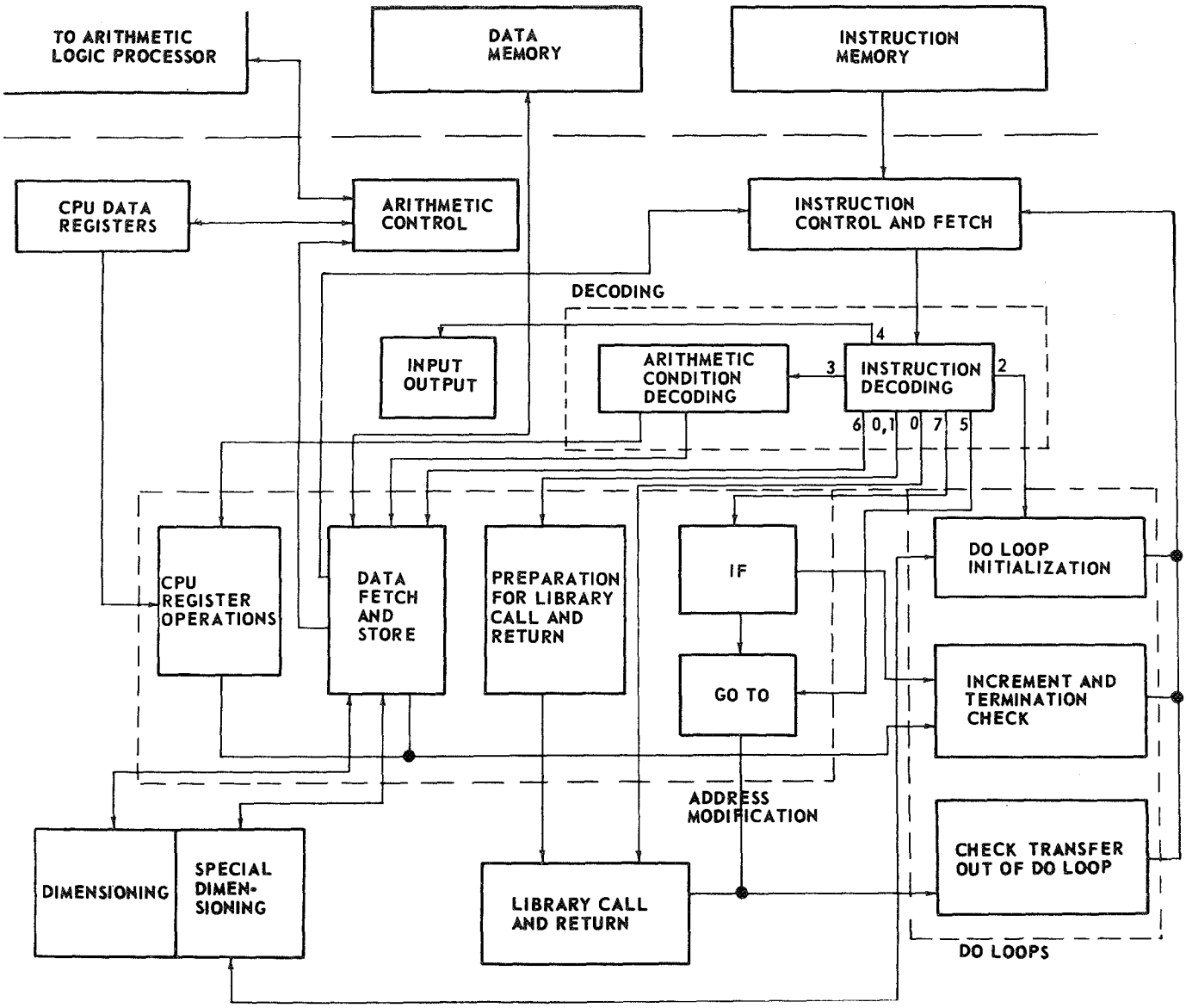


Figure 3. FLP flow diagram.

All operations were coded in the most concise form by taking advantage of some high frequency occurrences. These special cases offering maximum opportunity for shortening references to data are detected and specially treated by the processor in order to compress instruction memory.

The arithmetic instruction type will be used as one example to illustrate this method. Seven special cases of arithmetic operations, which are expected to occur frequently enough to be exploited for shortening memory references, are discussed below.

If the operand of an arithmetic operation is an integer, its value can be stored in the next consecutive instruction word, or if its value is less than 2^8, it can be stored in the ADINFO field of the same instruction. Some operands are intermediate results of previous arithmetic operations, which reside in one of the 32 internal CPU registers. These operands can be referenced in the CPU registers by the ADINFO field of the calling operation instruction. Frequently operands can be arranged in consecutive memory locations. Whenever this is possible, data of such a string can be obtained by retaining the address of the first

data item and incrementing this address by one for each consecutive operation of this arithmetic string.

Arithmetic operations dealing with dimensioned variables are treated as three different cases. In the general case (seven dimensions are permissible in this language processor implementation), the address of the first word in the array is contained in the instruction word immediately following the calling operation instruction. Subscripts are referenced in the ADINFO field of this second instruction word. If the values of the subscripts do not reside in internal CPU registers, an additional instruction word refers to their memory locations. For variables in a dimension greater than two, the data address will be computed by software algorithms. For one- or two-dimensional variables, data addresses are computed by special hardware routines using subscript values stored in internal registers. Only one subsequent instruction word, locating the first word of the array, is required.

The above special cases are differentiated by the preprocessor which relates this information to the FLP through the 3-bit ADD field of the arithmetic instruction.

A similar approach was taken in the case of the DO loop statement. The DO loop variables of this statement have the following standard meanings:

m1: starting value of the loop index

m2: limiting value of the index

m3: increment by which the index is modified

Special values of these parameters occur frequently and this fact can be utilized as follows to save bits and time when referencing memory.

If the starting value of the index (m1) is either one or two, this value can be noted in the 1-bit m1 field of the DO instruction. If the maximal value of the index (m2) is less than 32, it will be stored in the 5-bit m2 field of the DO instruction word. Whenever m1 and m2 do not fall within the above range of favorable values, an additional instruction word is needed for references to memory locations containing the parameter. It can, however, be assumed that these general cases will not occur too frequently.

As with the arithmetic operations, the preprocessor discriminates among the combinations of m1 and m2 for the general or the special cases and encodes this information in the 2-bit DOFLAG field of the DO instruction.

Since DO loop indices are most frequently incremented in steps of 1 (m3 = 1), a special m3 field in the DO instruction format identifies this case. In the general case of $m3 \neq 1$, additional instruction space is required.

The value of the last address of the DO loop can be encoded in shorter form by measuring its numerical distance from the beginning address of the loop. If the distance is less than 63, it can be referenced in the 6-bit last address field of the DO instruction, and the absolute address can be calculated from this information. The general case of a distance larger than 63 is indicated by zeros in this field and the value is carried in an additional instruction word.

DO loops can be nested within DO loops. In this implementation, the information concerning the first seven levels of nested DO loops are stored in a stack of seven special registers. An adjunct 3-bit register designates the level of the currently active loop.

Similar techniques exemplified by the arithmetic and DO loop instructions analyzed above were employed in handling the remaining FLP instructions. In summary, these techniques take advantage of high frequency occurrences to develop special case algorithms that allow more concise coding than a general approach covering all cases. High incidence of such opportunities appears in the location or arrangement of data, in the constraints imposed upon the length of data string addresses, and in the definition of statement parameters. Memory references are also reduced by storing intermediate results in a special set of internal CPU registers whenever possible.

In general, the techniques employed require that additional information regarding any special exploitable characteristics of the operation be part of the instruction word. The additional bits needed to signify these characteristics are expected to be highly repaid through the total memory savings achieved by this method. Also, the large number

of variations in the instruction word format imply a complex execution logic. This situation demands a highly sophisticated preprocessor as an intermediate step to keep the size of the logic hardware within reasonable bounds.

In the following section the economical implications of these techniques will be evaluated in terms of memory savings. Classes of computer systems and applications for which these techniques appear to be advantageous will be explored.

EVALUATION

Having functionally designed the FLP and defined the rules for the preprocessor program, we then evaluated the proposed approach to memory reduction by quantitative approximation.

Estimates were made of memory savings and of the added effort in hardware logic based on costs expected in the mid-1970's. These values were entered into a tradeoff equation that evaluates the relative influence of memory versus logic hardware. Four different missions are presented as examples of the memory-logic tradeoff evaluation.

For obtaining an estimate of memory savings, several programs were manually processed according to the rules established for the preprocessor. The programs were selected on the basis of their spaceborne application characteristics such as guidance and navigation programs. Manual preprocessing assembled and compiled the statements and packed the resulting instructions into the required FLP format. I/O instructions were omitted. A count of the FLP formatted words represents the FLP memory requirements of the programs.

The same programs were compiled on the IBM 7094. Code generated by I/O operations was deleted, and memory allocated for data storage was not taken into account. The number of words generated represents the comparative memory requirements for instruction storage on a computer executing machine language.

The ratio of the storage requirements for the machine language code over the FLP code, which is called compression ratio, resulted in an average of approximately 4:1 for the programs compared.

The estimate of the additional hardware cost was only concerned with the logic of the FLP that would have to be added to a conventional computer. Possible savings in the original computer control section, which might occur because of performance of similar functions in the FLP, were not considered. These concessions tend to produce conservative evaluation results. For purposes of this study, the design of the FLP was limited to the definition of the various registers and the associated information flow. The logic requirements were estimated by multiplying the number of register flip-flops by the ratio of logic gates to flip-flops usually found in computers. This ratio was obtained by examining several computers and was found to vary between the values 20:1 and 40:1. The lower case, 20, was used to calculate the number of logic gates required, because the FLP is a device using many more registers than is normal in computers. There are 2270 flip-flops employed in registers of the FLP, consequently the additional number of gates required was calculated to be 45 000 or less than 50 000. Since this amount of logic is common in medium to large scale computers, 50 000 gates is considered an upper limit to the size of the FLP.

For the tradeoff analysis, a memory compression ratio of 4:1 and a corresponding hardware estimate of 50 000 gates were used to arrive at a comparable cost figure. The total cost differential (C) between a computer using a FORTRAN Language Processor and one without this addition consists of differentials in manufacturing and development cost (M), in direct power cost (P), and in launch cost (L). The following equation expresses the generalized cost (C) for a total of n computers manufactured and one launched.

$$C = nM + P + L \qquad (1)$$

The manufacturing and development cost differential (M) contains three terms. The first term represents memory cost saving, the second term represents manufacturing cost of the additional logic hardware, and the third term represents the LSI chip development cost.

$$M = (s \cdot b \cdot r) - (g \cdot cg \cdot r) - \frac{g \cdot d \cdot cc \cdot co}{n}$$

$$= \quad m1 \quad -m2 \quad -m3 \ , \qquad (2)$$

where

M = net cost savings realized (memory reduction versus added logic),

s = size of memory saved (bits),

b = bit cost ($),

r = cost factor because of reliability requirements,

g = number of gates added,

d = chips per gate packing density,

cg = manufacturing cost per gate ($),

cc = development cost per chip ($),

co = fraction of total chips requiring new development,

and

n = number of computers to be manufactured.

Cost savings as a result of reduced power consumption and launch weight follow from equations (3) and (4).

$$P = (pm - pl) \cdot cw \qquad (3)$$

$$L = (wm - wl)1 + (pm - pl)ww \cdot 1 \quad , \qquad (4)$$

where

P = power consumption cost savings ($),

L = launch weight cost savings ($),

wm = weight of saved memory,

wl = weight of FLP logic,

pm = power saved because of memory reduction (W),

pl = power penalty because of the added logic hardware (W),

cw = cost per watt ($/W) ,

ww = weight per watt (lb/W)

and

l = cost to launch 1 pound ($/lb).

As a first example for the tradeoff evaluation expressed by equation (1), consider a computer aboard an Earth-orbiting space station in the last half of the next decade. This computer controls a portion of the 300 experiments aboard and processes their output data, a task which may vary from simple reformatting of data to fairly complex pattern recognition. It monitors spacecraft subsystems, and predicts and diagnoses failures. Data management and transmission are important functions of the computer which also serves as the center of command and control activities and is the heart of the guidance, navigation, and attitude control systems.

The memory requirements for this mission were estimated to be 260 000 38-bit words. Sixty-thousand words were allocated for storing the executive program, data, and work area leaving 200 000 words available for instruction storage. The instruction storage compression ratio of 4:1 expected with the FLP approach indicates a memory savings of 150 000 words of 5.7 million bits.

The other values needed for solving the tradeoff equations (1) through (4) were obtained from the literature [5-7] and updated with current information obtained from component vendors. The following table lists these values: (E means the exponent of the base 10, as used in FORTRAN notation).

s	memory saved	5.7 E6 (bits)
b	bit cost	3 E-2 ($)
r	cost factor because of higher reliability	3
g	number of gates	50 000
cg	cost per gate	1E-1 ($)
d	chips per gate	1.5 E-3
cc	development cost per chip	2 E4 ($)
co	fraction of new chips	3 E-1
n	number of computers	n

Figure 4 shows the plot of the three terms of equation (2). Manufacturing cost is represented by m2, and m3 represents development costs. The term representing the savings through memory reduction, m1, is shown as a function of the reduced bit size. The manufacturing cost ($15 000 each) represented by m2 was calculated for a production run of five computers: the flight computer, another

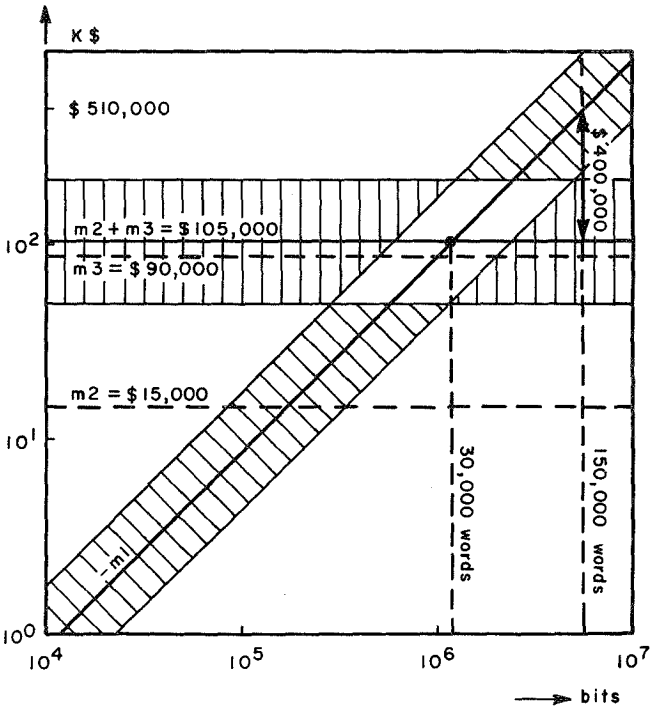


Figure 4. Tradeoff equation (2), small quantity (n = 5), high reliability (r = 3).

for testing, two for program development, and one backup computer. The development cost, m3, calculated as $90 000 per computer is nearly one order of magnitude larger than m2, the manufacturing costs ($15 000) of the FLP. The combined chip development and manufacturing cost (m2 + m3) equals $105 000, which differs from m1 by $410 000 for a memory savings of 150 000 words. Five computers could therefore be produced at a savings of more than 2 million dollars ($2 050 000).

Figure 4 also shows that the costs for logic and memory break even at a memory size of 1 million bits. For larger memories, the FLP approach should be economically superior.

To compensate for the uncertainty of the values, the plots of the m1 term and of the combined m2 + m3 term were represented by bands whose limits were defined by twice and one-half the calculated values. Figure 4 shows that even in the worst case (the lower boundary of m1 and the upper boundary of m3) the tradeoff is in favor of the FLP approach for memories of over 4 million bits.

The preceding findings were applied to a production run of 50 of the same computer for possible use in military ground or airborne applications. The chip development costs, m3, distributed over the 50 computers were reduced to $9000 per unit. The combined chip development and manufacturing cost, m2 + m3, per unit totalled $24 000. This indicates an advantage for the FLP approach when memory savings are 270 000 bits (7200 words) or more (Fig. 5).

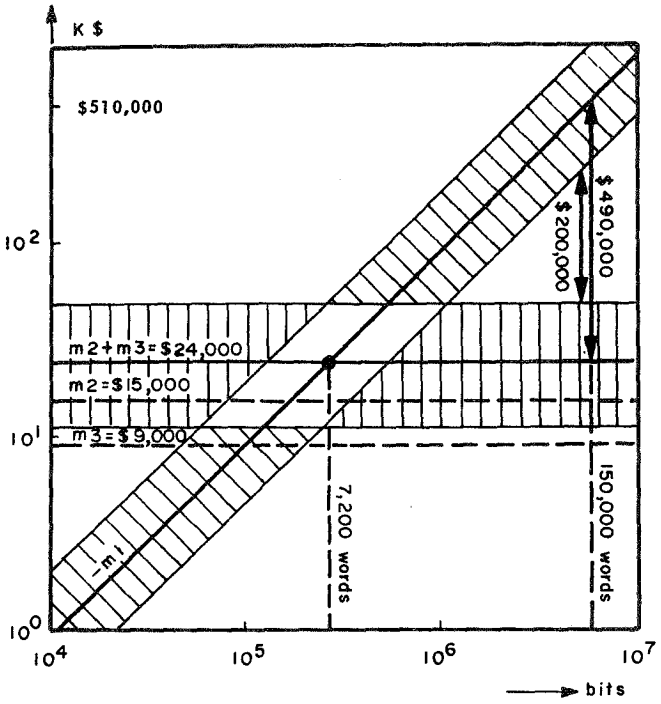


Figure 5. Tradeoff equation (2), large quantity (n = 50), high reliability (r = 3).

For commercial computers (Fig. 6, r = 1), with chip development and manufacturing costs (m2 + m3) of $14 000, a break-even point appears at 500 000 bits or 13 000 words. Consequently, a memory reduction of 150 000 words in a commercial version of the FLP would save $146 000 per computer.

Assuming the memory compression ratio of 4:1, it is concluded that military computers with more than 10 000 words and commercial computers with more than 17 000 words required for instruction storage can benefit from a high level language processor.

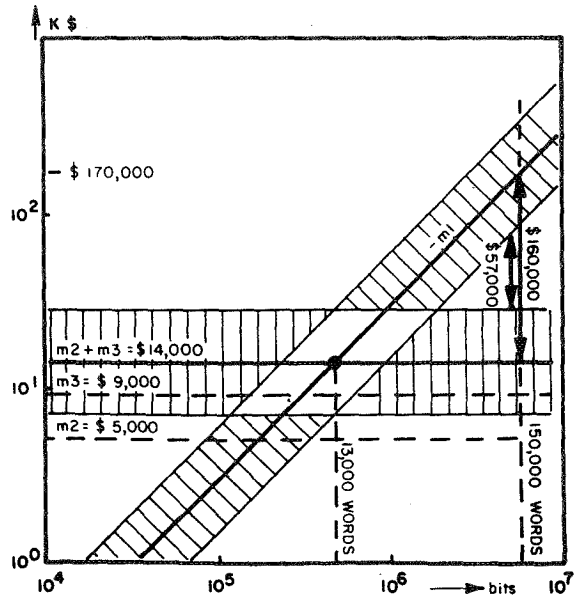


Figure 6. Tradeoff equation (2), large quantity (n = 50), commercial reliability (r = 1).

For spaceborne computers, the cost of the power supply and the launch cost of the computer may constitute a substantial portion of the cost of the total computer system.

The cost savings resulting from the reduced power requirements (P) of the FLP design were calculated by equation (3), P = (pm - pl) · cw, using the following values:

pm, memory reduction of 150 000 words saves: 580 W

pl, the added hardware logic consumes: 50 W

cw, cost per watt (solar cells): 200 $/W,

then

P = $106 000.

Launch cost savings (L) attributable to the FLP design were calculated by equation (4), L = (wm - wl) + (pm - pl) ww · l, using the following values:

wm, weight of saved memory: 230 lbs

we, weight of added logic: << 230 lbs

(pm - pl), reduced power requirements: 530 W

ww, weight per watt: 0.16 lbs/W

l, cost of launching one pound: 1000 $/lbs

then

L = $230 000 + $85 000

L = $315 000 .

(The weight of added logic hardware was considered negligible when compared with the weight of the memory saved.)

Equation (4) indicates that the weight of a computer with a 260 000 word memory could be reduced 315 pounds by adopting the FLP approach (230 pounds could be saved through memory reduction and 85 pounds through reduced power requirements).

The above launch cost savings of $315 000 is for an Earth-orbiting mission where current launch costs average about $1000 a pound. These savings appear insignificant when compared to the 2 million dollar reduction in manufacturing costs or when compared to the cost of the total mission.

Launch costs of $20 000 per pound are predicted for interplanetary missions of the mid-1970's. In this case, launch cost savings because of weight reduction of the FLP approach will exceed 6 million dollars in addition to the 2 million dollars saved in manufacturing costs. The ultimate payoff of this savings is the opportunity it affords for payload sophistication, thereby increasing the scientific value of the mission.

However encouraging this result is for interplanetary missions, the final analysis of a spaceborne computer system must include its reliability. This task must be postponed until sufficient data on LSI reliability is available.

CONCLUSIONS

This paper attempts to answer the question of whether the traditional allocation of functions between logic and memory will change under the impact of new technology or extraordinary performance requirements such as those imposed by space flight. The conclusions reached as a result of this investigation stand on projections of manufacturing costs in the mid-1970's, on a particular language, compressed instruction coding, and interpretive logic. Despite wide variances, tolerated in all numerical values used in this paper, the authors contend that some positive conclusions were reached regarding the benefits of the higher level language processor approach.

From the economical point of view, the example worked suggests advantages for all four cases whenever the memory required for the application was sufficiently large. For a computer with a memory of 260 000 words, it was considered feasible to reduce its size to 110 000 words through the approach offered. This reduction of 150 000 words implied a manufacturing cost savings of about $400 000 for computers demanding high reliability when produced in a batch of 5. For computers manufactured in batches of 50, a break-even point between the additional logic hardware and the saved memory occurred around 13 000 words for commercial computers and at around 7000 words for high reliability hardware. Since these represent rather small memories, the addition of a higher level language processor appears to be advantageous even in commercial quality hardware. The reduction in manufacturing costs for such computers containing larger memories appears to be the significant factor (some $100 000). For interplanetary missions, however, the reduction in weight and volume proved more important. A key consideration for these long duration missions is the impact the additional processor would have on system reliability.

The economies of the higher level language processor approach demonstrated in this paper suggest that additional benefits can be derived through improvements in the scheme. Investigations of languages other than FORTRAN may reveal even greater opportunities for code compression. Some excellent candidates should be found among real-time control languages and algebraic types capable of processing arrays of data or executing complex algorithms by single instruction [8-10]. Problem-oriented languages, which we ranked one level above procedure-oriented languages in code compacting qualities, seem to offer great opportunities for code compressions. An investigation of data structures should uncover additional memory saving possibilities. A more detailed analysis than the one done for this paper should investigate individual features implemented in logic hardware and evaluate the tradeoff involved.

Further improvement of the tradeoff results seems to be possible through standardizing some of the logic components, especially heavily used stacks and counters. Employment of microprogramming schemes could replace specialized logic by standardized microcontrol hardware. This approach may improve the reliability of the system through the interchangeability of the microprogrammed control devices.

Besides cost, the computer organization implied by the higher level language approach can influence other factors of computer architecture. It can for example, offer a solution to the program segmentation problems encountered in multilevel memory schemes (paging, cache technique [11-13]), because complete statements are transferred to lower level processors, where the equivalent of small segments are generated. Furthermore, since the higher level statement is not compiled into memory machine steps, extensive look-ahead features can be more easily implemented.

Future computers also face a bandwidth problem in the internal communication between computer subsystems. This is especially serious in the case of distributed logic computers. The instruction stream through such computers can be greatly reduced through the employment of higher level language processors.

In conclusion, this paper indicates that employment of more comprehensive instructions or complete higher level languages is desirable not only because of their inherent speed and programming advantages, but because they also provide economical advantages by virtue of their memory saving properties.

REFERENCES

1. Spaceborne Multiprocessing Seminar. NASA Electronics Research Center, Cambridge, Mass., October 1966.

2. Bashkow, T. R.: A Sequential Circuit for Algebraic Statement. IEEE Transactions on Digital Computers, April 1964.

3. Lawson, Harold W.: Programming Language Oriented Instruction Streams. IEEE Transactions on Digital Computers, vol. C17, no. 5, May 1966.

4. Burroughs 6500/7500 Information Processing System Characteristics Manual. September 1968.

5. Wainer, R. M.: Comparing MOS and Bipolar Integrated Circuits. IEEE Spectrum, June 1967.

6. Einhorn, Richard N.: LSI Improves Computer Memory Bit by Bit. Electronic Design, April 1, 1968.

7. Brown, D. W.; and Burkhordt, D. L.: The Computer Memory Market. Computer and Automation, January 1969.

8. Seitz, Robert N.; Wood, Lawrence H.; and Ely, Charles A.: AMTRAN: Automatic Mathematical Translation. Interactive Systems for Experimental Applied Mathematics, Proceedings of the Association for Computing Machinery Inc. Symposium, Washington, D. C., August 1967, p. 44.

9. Falkoff, A. D.; and Iverson, K. E.: The APL/360 Terminal System. Interactive Systems for Experimental Applied Mathematics, Proceedings of the Association for Computing Machinery Inc. Symposium, Washington, D. C., August 1967, p. 22.

10. Symes, Lawrence, R.; and Roman, Roger V.: Structure of a Language for a Numerical Analysis Problem Solving System. Interactive Systems for Experimental Applied Mathematics, Proceedings of the Association for Computing Machinery Inc. Symposium, Washington, D. C., August 1967, p. 67.

11. Wilkes, M. V.: Slave Memories and Dynamic Storage Allocation. IEEE Transactions on Digital Computers, April 1965.

12. Conti, C. J.; Gibson, D. H.; and Petowski, S. H.: Structural Aspects of System/360 Model 85. IBM Systems Journal, vol. 7, no. 1, 1968.

13. Liptay, J. S.: The Model 85 Buffer Storage. IBM Systems Journal, vol. 7, no. 1, 1968.

Page intentionally left blank

FUNCTIONAL DESIGN CONSIDERATIONS FOR AN EXECUTIVE SYSTEM FOR A GENERAL PURPOSE SPACEBORNE COMPUTER

By

J. R. Kennedy

ABSTRACT

An overview to spaceborne computer requirements for general purpose application in advanced missions is discussed in this paper. An approach to a functional design for an executive system that satisfies these gross requirements is presented from an organizational viewpoint with no consideration given to implementation. The level of detail brings out the feasibility aspects of communications, task routing, application command and control, organizational modularity, and mission schedule interpretation.

SUMMARY

Spaceborne computer usage is envisioned as a broad application of computational facilities in the form of a centralized (complex of) computer(s). The requirements associated with this concept are summarized and shown to be extensive. Detailed requirements have not yet been developed. However, based on a heuristic functional classification of applications, an overview to a functional design approach for an executive routine is offered for consideration. It is inherently reliable and flexible, and allows for multiple implementation schemes that can be comparatively analyzed for their relative merits.

INTRODUCTION

The concept of digital computers for spacecraft guidance is well established [1] and has proven itself in manned flight [2] on numerous occasions during the latter phases of the Apollo project. Furthermore, the desirability of using digital computers for a wide variety of additional functions including monitoring and adaptive control of telemetry systems and monitoring vehicle systems for malfunction detection is an accepted goal [3]. In fact, it is clear that the desire for higher capability outstrips the ability of technology to provide the facilities [4]; that is, large scale integration (LSI) has not yet become a reality. However, major research efforts, partially funded by NASA and the Air Force, have been underway, notably at the Autonetics Division of North American Rockwell Corporation [5], the Federal Systems Division of UNIVAC [6], and the Hamilton Standard Division of United Aircraft Corporation [7]. There is therefore a large amount of work being done in an effort to determine modular approaches to the design of ultrareliable general purpose digital computers. This report outlines a functional, rather than a computer hardware control-oriented [8, 9] approach to the design of an operating system (executive routine) for the utilization of future spaceborne general purpose computers. The major objection to a functional approach is that it seldom considers detailed hardware aspects, such as interrupt system control. It is felt that most of the details of hardware mechanization can and should be submerged below the level of normal programming, including that for the operating system, by intelligent use of microprogrammed control logic. This is not necessarily a popular opinion, though it has been suggested elsewhere [6, 10]. It is true that the problems associated with such things as interrupt processing must be solved on future spaceborne computers. However, it is suggested that the problems are hardware oriented and should — and will — be worked out in the future by a new, hybrid breed of system specialist; i.e., the firmware programmer/designer. On the other hand, there is much to be gained from a functional design approach as will be seen in the considerations to follow. Several of the concepts advocated in this report lean heavily toward total (but alterable) computer control of future space missions, from countdown to mission completion.

FUNCTIONAL REQUIREMENTS

Functional requirements for the spaceborne computer of the near future have been well outlined

elsewhere and will not be amended here except to point out a few of the more important requirements and to summarize.

Mission Oriented Requirements

FLEXIBILITY

Based on historical evidence related to both ground-based and spaceborne computer development and implementation, the attainment of general purpose capabilities can be expected to cost large sums of time and money. For this reason, primarily, the systems that evolve should be designed around the concept of mission independence (flexibility). This will enable an amortization over many missions with an expected lower cost than a single-mission design approach. It should be possible, for example, to reconfigure both the hardware and software with a minimum of redesign so as to accommodate a variety of missions equally well — from manned or unmanned Earth-orbital missions to lunar landing, planet flyby, deep space probe, etc. It is expected that this inherent flexibility will also improve mission reliability by an increase in the system's capability to perform in a state of "graceful degradation"; many of the features required for true mission independence are the same as those needed for performance under conditions of failure.

RELIABILITY

This term applies here in all of its software and hardware aspects, but particularly in its relation to crew performance. The system should include facilities for crew supportive functions such as training, testing, and morale sustenance on long duration manned flight or orbits. For instance, it has become obvious that even a short mission, such as the 74-hour translunar trip, is extremely boring and requires supplementary crew entertainment and training to enrich the long, idle hours.

SUMMARY

The many functional requirements as outlined by Gruman and Schaenman [3] and suggested by Black [11] are summarized here to emphasize the extent to which the application of computing power is envisioned. This is the framework on which the executive design is based.

It is intended that the general purpose spaceborne computer provide services for test and checkout, astronaut-computer interaction, data processing, vehicle systems, experiments, and communications systems.

In the area of experiments, it is necessary that the computer function as a process control subsystem for the purpose of sensor monitoring, data logging, control signal calculations and generation, equipment calibration, experiment initialization, and data analysis [12, 13].

For effective astronaut services the operating system must include provisions for the control of astronaut display/keyboard-entry facilities for astronaut-computer interaction. It should also provide a "desk calculator" facility and an astronaut diary [log]. In general it must be the communications medium whereby an astronaut-scientist can direct the progress of experiments and command various vehicle subsystems to perform in arbitrary or predetermined sequences. Other examples of astronaut services that will be provided are command generation to enable the control of guidance equipment and checkout capabilities, malfunction analysis assistance, and the ability to review and edit a preplanned mission schedule.

Services in support of test and checkout functions include the generation of system stimuli, the acquisition and analysis of system responses, malfunction detection, monitoring, and analysis, confidence testing, and diagnostic control. This capability must interface smoothly with the astronaut interaction facility to allow the astronaut to direct various testing and checkout procedures and to assist him in the possible replacement of malfunctioning line replacable units [14].

In support of vehicle systems, the operating system will provide facilities for vehicle system monitoring, control, and calibration. In addition to the normal calculations for steering, navigation, propulsion systems control, control moment gyro control, and life support system monitoring, the vehicle system capability within the operating system will allow for power management and control. This latter function will pay particular attention to computer system dynamic power requirements to ensure minimum power consumption as a function of computing requirements. This task can be performed

by switching power off or to standby on computer system modules that are not required because of reduced demands.

Communications system support facilities will include the ability to perform data buffer management, provide for packet transmission, and provide for data compression and expansion as a means of bandwidth (power) control. Certain standard data compression algorithms such as those required for reducing or eliminating redundancy and smoothing by polynomial fitting will be provided as standard procedures for all telemetry data. Also, the communications capability will allow for dynamic priority and tolerance control over data channels. Various transmission diagnostics will be available along with the ability to interpret the results of system performance. A capability for controlling telemetry format on a dynamic, real-time basis will be provided to further adapt the telemetry system to changing demands [15].

Certain functions related to the existence of the general purpose computer will be required in the form of executive control services. Those that are directly related to the spacecraft itself include mission schedule interrogation and interpretation, event recognition and verification, mission schedule editing, time keeping, and priority conflict resolution.

Data processing capabilities must be provided to enable the astronaut-scientist to perform a variety of data analysis and manipulation functions. For instance, it will be desirable to convolve one block of data points with another, perhaps for visual image pattern recognition [16] to assist in star-pattern detection. Fourier transforms and inverses will be necessary for experimental data power and frequency spectral analysis. Conventional operations such as block additions, averaging, correlation, etc. will also be provided. A comprehensive set of operational capabilities [17] is required to enable effective onboard data evaluation; careful data analysis will most certainly reduce telemetry requirements and improve astronaut decision capabilities.

OPERATING SYSTEM FUNCTIONAL DESCRIPTION

This preliminary document is an interim report covering a basic design approach for a general purpose spaceborne digital computer operating system. The intent of the document is to set forth the gross organization of the operating system and to define the various functional building blocks making up the system. Since the effort expended up to the time of this report was concerned with a conceptual approach as opposed to a detailed implementation plan, the design as outlined herein is somewhat independent of any specific computer configuration. However, certain assumptions simplify discussion and are listed here.

Configuration Assumptions

In most cases, the actual omission of assumed features from any feasible future target machine will simplify the task of the operating system, but in no case will an omission compromise the design concept.

MULTIPLE PROCESSORS

It is assumed that the hardware will be made up of several processors which can execute in parallel. No assumption is made about the organization of these processors. They may each be capable of performing identical tasks, but possibly not at the same level of performance, or they may have diverse functions such as input/output control, high level language execution [18], arithmetic processing, etc. Furthermore, no assumption is made as to what tasks are being performed by the various processors. It is assumed, however, that there is a hardware mechanism (special registers) for operating system control of the assignment of tasks to the various processors according to arbitrary decision criteria. In particular, it is assumed that the operating system, while being interpreted (executed) by one processor, can cause another processor to cease its current activity and commence to interpret (take over alone or simultaneously with another processor) portions of the operating system code. The effect of the above functions is that the operating system can arbitrarily schedule the work of any processor, including the ability to switch portions of itself from one processor to another or cause parallel, possibly synchronous, interpretation of itself by several, possibly redundant, processors. The obvious reasons for these assumptions is to allow for processor scheduling, including multiprogramming of processors, and to increase overall reliability in case of certain processor malfunctions.

MASS STORAGE

Second, it is assumed that the hardware system will contain mass storage facilities for software program and data residence. It should be pointed out at this time that the class of programs residing on this mass storage is noncritical in the multiple sense that: (1) they are not time-critical and (2) they are not mission-critical. The procedures making up this class can all be reconstructed from at least two sources: ground-based via telecommunications and spaceborne via astronaut. Those elements of the system that are time-critical or mission-critical are assumed to reside permanently in, possibly, a nondestructive read-out read-only-memory (ROM), although they could reside in a more conventional memory with less reliability. The reason for this assumption is to reduce cost.

INTEGRATED SYSTEMS

Finally, it is assumed that all checkout and test equipment, astronaut-computer communications equipment, vehicle (controllable) systems, experiment packages and telemetry equipment are accessible from the central spaceborne computer and that the defined function of this spaceborne computer is to monitor, control, and communicate with any and all of this equipment essentially on a demand basis in a virtually asynchronous mode of operation. This assumption is self-explanatory.

Operating System Functional Organization

As mentioned before, the operating system will have the broadly expressed functions of monitoring, controlling, scheduling, and communicating with all spacecraft systems, including communications with ground-based systems and other spacecraft (such as logistics systems, relay satellites, etc.) that are considered to be logical extensions of the spacecraft's own systems. In order for these functions to be performed in a reasonably coherent way, the overall organization of the operating system is structured in a modular form that allows for the deletion or inclusion of certain subsections, provided that the resulting system exhibits self-consistent integrity. For instance, although the system is envisioned to include a section that is capable of interacting with an astronaut-scientist, this section could be deleted in its entirety without any reprogramming for a nonmanned mission and with no effect on the remaining portions of the system, except insofar as the exclusion of an astronaut-scientist affects the mission. A section having to do with automatic test and checkout of spacecraft systems other than the central computer could be excluded at the expense of the reliability of those systems. It is equally feasible to envision that as missions become more complicated, other sections might be added although they may not be obvious at this time. There is an inherent advantage to a modular organization found also in the concept of graceful degradation. For instance, by associating an operational (usage) priority with each system module, the system becomes more adaptive [19].

As is shown in the following section, the operating system is composed of a set of monitors that function somewhat autonomously to perform the particular function for which they are best suited. These monitors depend on a system monitor, called the executive monitor, for common services such as intercommunications among monitors when this is required. The executive monitor serves as the operating system prime mover in the sense that it interrogates a coded mission schedule and translates the schedule elements into event occurrences and/or monitor tasks that drive all monitors to perform their function according to the mission schedule and for overriding its controlling sequence from external (ground-based) or internal (astronaut) sources. These will be discussed later at a more appropriate point.

In the following discussion, monitor functions are represented as self-contained capabilities for the sake of simplicity of presentation.

MONITOR STRUCTURE

Monitors are all structured in an identical form in the interest of simplicity and compatibility (Fig. 1). A uniform structure results in simpler program verification during implementation in addition to an acceleration of the development effort involving coding because of similarities in programming techniques and interfacing between monitors. The major benefit of a uniform structure, however, takes the form of simplified flow of control within the operating system and among the various monitors. This will lead to a formulation of more natural algorithms for the development of large general purpose sections of the system. Furthermore, documentation will be simpler and more understandable, another feature related strongly to program verification and development rate.

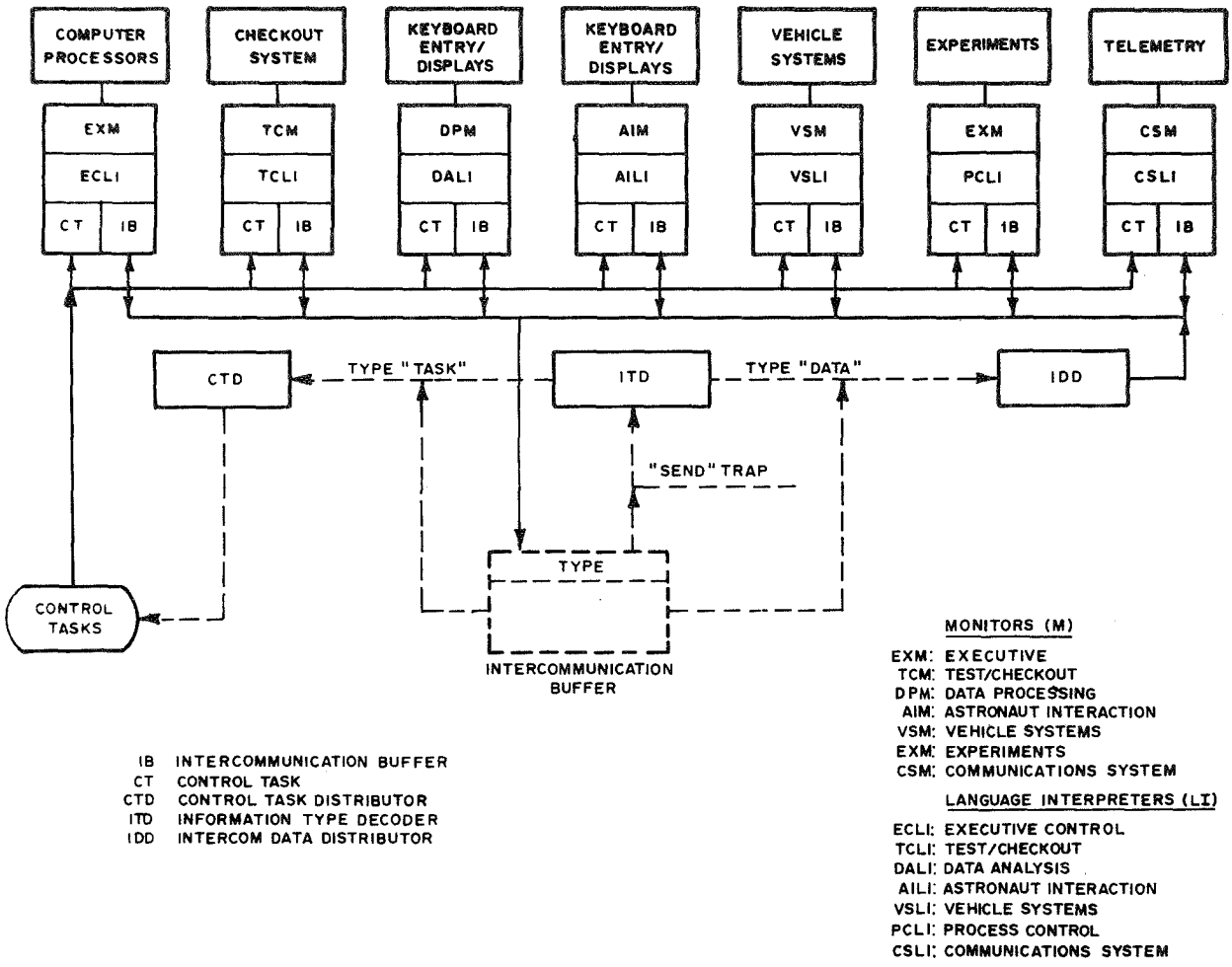


Figure 1. Control of system monitors and monitor intercommunications.

General Concept. Each monitor contains a control language interpreter that examines a sequence of instructions (commands or procedures) and performs the indicated actions in order to fulfill its function. In addition to a control language interpreter, each monitor contains a set of procedures that perform services peculiar to that monitor. These services are the heart of the computation portion of each monitor and as such contain all the detailed instructions to allow the monitor to perform its primary function. The final logical portion of each monitor is an intermonitor communications package. The purpose of this collection of routines within each monitor is to allow the monitor to send and receive data to and from other monitors. These data normally take the form of monitor state parameters that are communicated back and forth on a dynamic basis between subsections of the spaceborne system to enable logical communications among the various pieces of hardware and software.

Consider the following normal mode of operation for a monitor. A control task is routed to a monitor based on the occurrence of some event. The event could be mission scheduled, astronaut provoked, or internally generated. The control task is interpretively executed by the monitor control task interpreter with the result that execution will cause the performance of certain service functions as specified by the control task. In this way a control task normally establishes the state of a monitor (configures the monitor), thereby directing it to perform its function in a certain way. It is possible for the monitor service procedures to collect and transmit various parameters via the intermonitor communications mechanism for use by other monitors. It is further possible for a monitor to receive parameters that will be used in the performance of its service functions. The process of sending and receiving via intermonitor facilities is directed by control tasks. Depending on the nature

of a particular control task, it may be replaced at some arbitrary time by another task that could reconfigure the mode of operation of a particular monitor by changing its relation to external elements. Whether a particular active task can be replaced by another task at a given time will be determined by the currently active task thus providing the facility for locking-out external changes in the monitor configuration depending on the state of the monitor.

Control Interpreter. The language used to control a particular monitor is unique to that monitor and is problem-oriented in the sense that it is tailored to the requirements for command of the monitor as they relate to the unique functions for which the monitor is responsible. For instance, the control language for the test/checkout monitor is an automatic checkout language [20] that will direct the service facilities of the test/checkout monitor in the functions of testing, calibrating, monitoring, etc. the systems properly interfaced to the computer. One or more of these control procedures might communicate with the astronaut through a test/checkout panel for manual control and intervention. Another important example of current day usage of languages that are interpretively executed is the commonly used job control language [21] for ground-based systems. These languages are normally expressed in the form of punched cards inserted at the beginning of program decks that are to be processed by third generation multi-programming systems. Job control language cards are interpreted by a service routine within the operating system and are used by a programmer to control the operations to be performed by the system. Examples of functions that can be controlled by the programmer in this way are language translation, program loading, mass storage allocation, file opening and closing, etc. An executive monitor within the spaceborne system will be directed in a similar manner by a command language.

It should be emphasized that these languages are interpreted, not compiled. Interpretation has many inherent advantages over compilation when the language is used as a controlling (command) medium instead of a computing medium as is normally the case. For instance, interpreters are far simpler to design, implement (code and verify), and modify than are compilers. Furthermore, when the languages are simple and direct, as are the controlling languages being considered here, execution of the control procedures expressed in these languages is generally faster than a compilation would be. However, speed of execution is not felt to be a critical factor in the case of procedures used simply for monitor control. In fact, interpretive techniques have been used quite successfully [1] in cases requiring speed and accuracy. Also, the use of interpretive control permits the retention of procedures in their source language form (compressed) thereby allowing for ease in editing to modify control procedures, a powerful flexibility feature that enhances the ability to cope with unexpected contingencies.

In addition to the advantages cited above, interpreters can be easily implemented as micro-programmed ROM control logic. This would greatly increase execution speed and overall system reliability. Furthermore, certain portions of an interpreter are common to all interpreters. An example of this commonality is found in the case of a procedure for scanning source language statements. This procedure can be programmed as a general algorithm which allows it to be used in an elegant and powerful way to scan arbitrary source code. In this way it can be shared among all of the interpreters possibly as a re-entrant routine in microprogrammed ROM for a large savings in memory.

Figure 2 shows a simplified flow chart for an interpreter to illustrate the relationships between control tasks, interpreter decoder, and control loop, and service routines. A hardware designer familiar with the concept of microprogramming will recognize immediately the striking similarity between this and a diagram for digital computer stored-logic control. This similarity is a natural one since the process of microprogrammed control is, in fact, an intepretive procedure. This likeness strongly suggests that the system interpreters be implemented as microprogrammed logic. If the control tasks can be structured with language elements that are syntactically homogeneous (stem from a common grammar), it will be possible to consider using a single interpreter to execute all control tasks. This would be true, of course, regardless of the implementation. A comparative analysis of the tradeoffs associated with each approach would have to be made to determine which scheme is best for a particular monitor.

Service Procedures. The performance of monitor services is implemented by a collection of logically independent blocks of procedures making up the computing core of a monitor. These blocks are

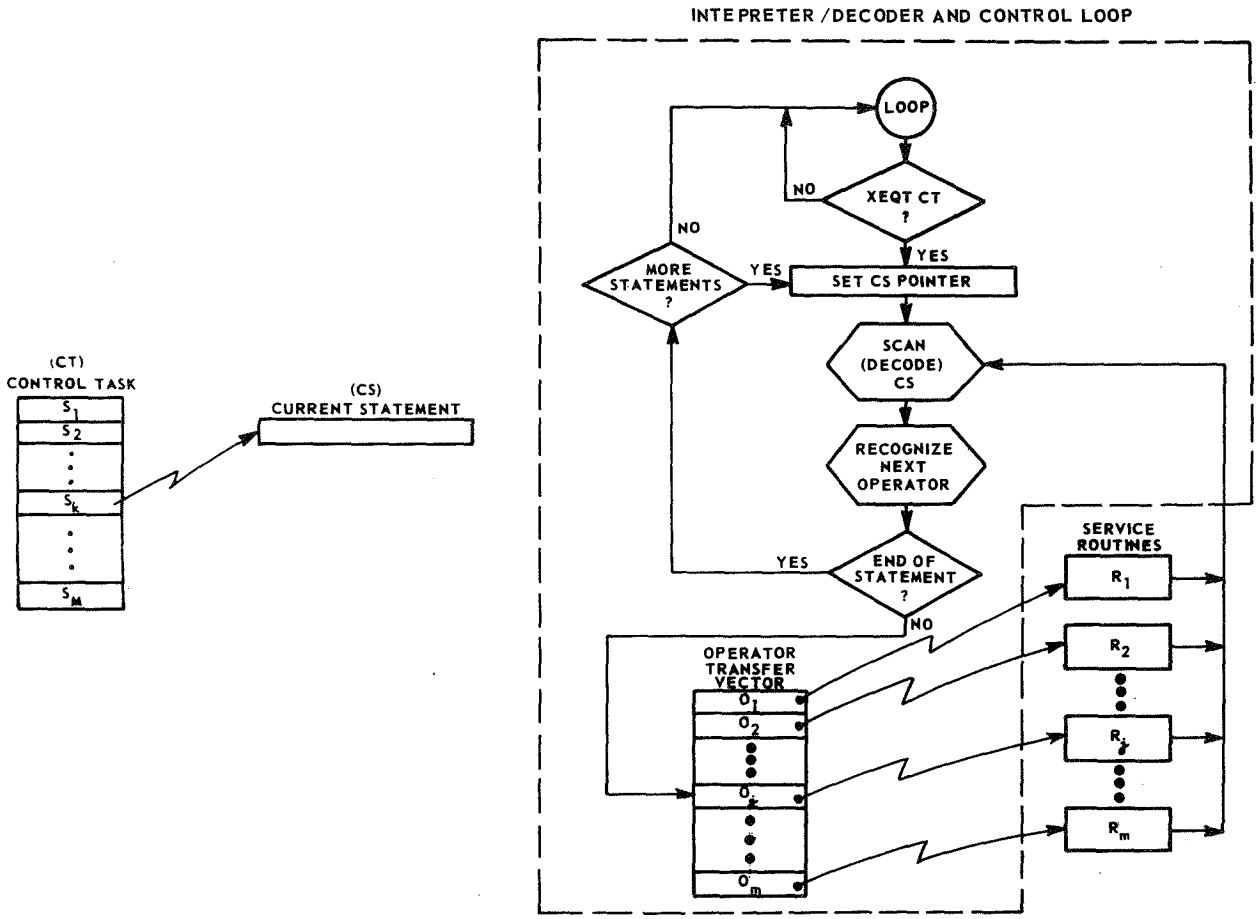


Figure 2. Simplified interpreter.

concatenated into sequences of logical steps by the interpretation of control tasks that specify the order in which procedural blocks are to be executed. The service procedures consist of routines to perform control task services and to provide utility and housekeeping capabilities within the monitor.

Those routines that perform control task services are a logical part of the interpreter and are connected to it by a transfer vector (or jump table). As the interpreter scans control task statements, it branches through the transfer vector to appropriate service procedures for execution of the statements. The housekeeping and utility routines perform incidental functions such as stack manipulations, input/output, storage allocation, etc.

As an example of specific service procedures, consider the vehicle systems monitor. To provide guidance control commands, service calculations will be performed by the monitor to obtain parameters for subsequent output to a control system to gimbal the propulsion engine for steering purposes. Other procedures will be required for coasting, tracking, burn sequences, position updating, etc. Another example is found in the experiments monitor where procedures for controlling experiments and gathering data (logging) are provided.

Intermonitor Communications. Information to be routed between monitors is formatted and encoded by the sender and decoded by the receiver. The routing of each buffer is handled by one of two service procedures within the executive monitor, the control task distributor (CTD) or the intercommunications data distributor (IDD). A buffer to be routed from a monitor is fetched from the sender by an information type decoder (ITD) routine within the executive monitor. One scheme for routing is adequate. It is referred to as SEND and is

discussed below. The fetching operation of the ITD is activated by a SEND request issued by the sender. The ITD examines the buffer to determine its type (task or data) and routes it to the proper distributor for transfer to the receiver (Fig. 3).

This mechanization is the software equivalent to a hardware data bus within a multimodule digital system. The advantage of this technique is, of course, that each monitor is logically isolated from the rest of the system by bussing routines. This allows a particular monitor to be removed from the system for mission adaptation without affecting the remaining portions of the system. (This is not true of the executive monitor, since its purpose is to integrate the other parts of the system.)

There are constraining implications that result from this approach. Each buffer must be formed as two logically separate parts: a head and tail. The head contains rigidly formatted information for use by the bussing routines in determining buffer length, receiver identification (address), sender identification, and buffer tail content type. Other coded information might also be included in the head but the above examples give the important items.

The tail contains variable, coded data that are the information to be transferred to the receiver. This information can be the "name" of a control task (in this case the buffer type is "task") to be routed to the receiver for subsequent interpretation, or the buffer tail can serve to route various data values (buffer is type "data") among the monitors. A specific example of control task routing might be the activation of a control task for the vehicle systems monitor to inhibit response to astronaut-generated steering commands. The test/checkout monitor might initiate this action upon detection of a malfunction in the engine gimbal system, or the data processing monitor could cause the test/checkout monitor to perform a specific data acquisition task for the purpose of subsequent data analysis by the astronaut-scientist. Another example of data routing is the transfer of a block of 100 sampled data values from the experiments monitor to the data processing monitor for correlation with another block of data.

SEND (Forced Buffers). The ability to allow a monitor to send an unrequested buffer is a forcing feature needed by all monitors. This communications mode requires that the sender format a standard buffer containing head and tail information as outlined previously. A SEND request making available a pointer[1] to the buffer is issued by the sender to the executive monitor. The information type decoder uses the pointer to examine the buffer for type and

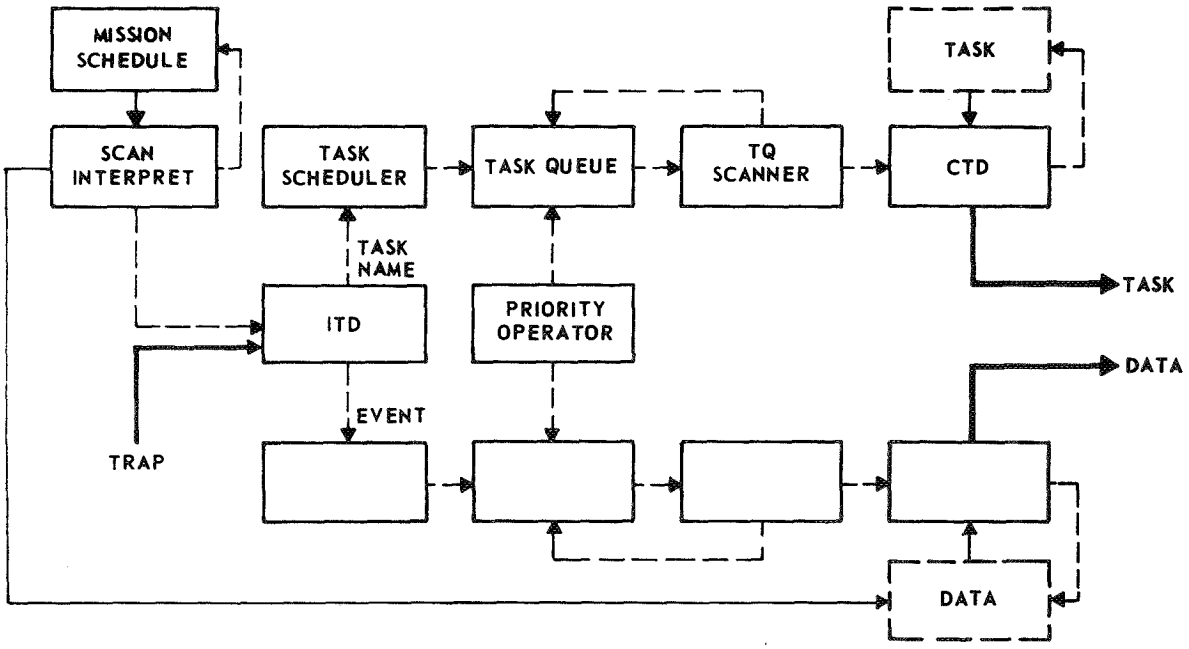


Figure 3. Monitor intercommunications and control task routing.

turns control of routing over to the appropriate distributor.

If the buffer type is data, the buffer is queued on a received data stack for the receiving monitor. This queueing is accomplished by the intercommunications data distributor[2]. When the receiving monitor removes the entry, buffer, from the stack, it then decodes the contents of the buffer tail for action. The tail can contain either pure data, in cases where there can be no possible ambiguity, or coded data. Coded data normally consists of a sequence of pairs (code, values). The first member of the pair is a unique code[3] known to both sender and receiver and represents the medium for dialog between the monitors. The second member is an array of data values (possibly nil) to be associated with the code. The array contains as its first element a count of the number of elements remaining in the array. See Figure 4 for an illustration of a typical buffer.

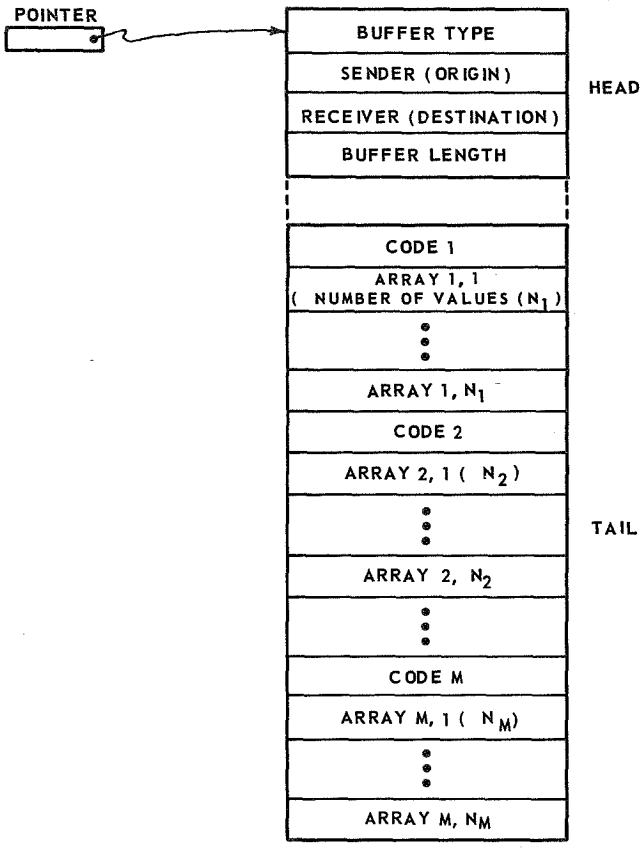


Figure 4. Intermonitor communications buffer format.

MISSION SCHEDULE

It was mentioned earlier that the mission schedule is envisioned as the system driver. Some general comments regarding the feasibility of this concept seem in order. The nearest related procedure[4] currently being employed that can be extrapolated into this concept is the semiautomatic Saturn V launch countdown. For this discussion, a brief review of launch procedures is in order.

The countdown sequencing is, of course, based on a countdown clock which gives decreasing time-to-launch (T). This time is used by a large number of test engineers as a reference baseline into a launch procedures handbook unique to each particular mission. Simply stated, as the clock cycles down, each test engineer or group of engineers looks ahead in the handbook to find the next test for which he is responsible. When the clock coincides with the time at which he is to conduct a test, he presses buttons on his test console (consisting of various

1. When a buffer pointer is relayed to another monitor, the buffer space in memory is automatically relinquished to the receiving monitor. Hardware could be designed to make this space inaccessible to the sending monitor.

2. This queue suggests the nature of driving each of the monitors (excepting the executive). That is, each monitor has an idle loop wherein it continually checks the status of various queues, one of which is the received data queue.

3. There is a set of system-oriented global codes used by all monitors to communicate functions that are common to all monitors. An example of a global code is the code for DATA REQUEST used by a monitor requesting data from another monitor.

4. The word procedure is a better choice than system since the process is actually sequenced by humans.

displays that enable him to visually validate test results) to direct the test(s).

The time required to perform and visually validate tests has been compensated for in the procedures handbook so that, as the clock cycles, each test is conducted such that there is plenty of time between the end of one test and the beginning of the next. This slack in the time-base is necessary of course because of the large standard deviation of the human test engineer from mean test time.

The significant point brought out above is that the launch countdown is actually sequenced (conducted) by humans according to a pre-recorded guide manual. Refer to Figure 5 and consider the role of the computer.

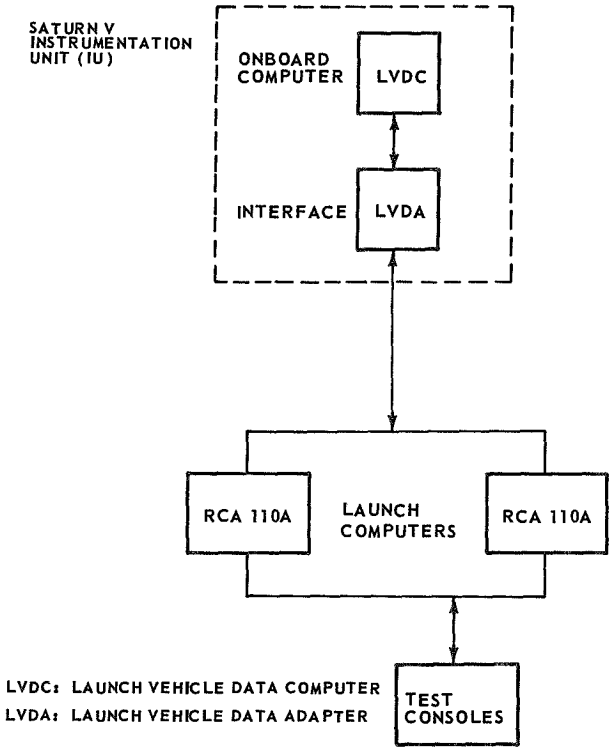


Figure 5. Launch complex computer configuration.

The launch computers contain programs coded to perform each test required during the countdown. In addition, they update the countdown time and monitor the Launch Vehicle Data Computer (LVDC) located in the instrumentation unit on the vehicle.

At time T-168 seconds, control and sequencing of the countdown are switched to an automatic launch sequencer program in the LVDC, and no test conductor intervention is allowed except to signal an abort.

Prior to T-168 seconds, the launch testing functions are controlled by test engineers through functionally oriented test consoles; the launch computers have no programmed knowledge of how to sequence the tests. In this respect, the launch computers play a passive role, as does the LVDC until T-168.

It is instructive, however, to consider a conceptual change that would give equivalent results but be more viable in an evolutionary sense. Suppose the guide manual was used in abbreviated form wherein it provided the name of tests to be performed as a function of countdown time and the address of the test console where the test is to be validated. This coded countdown schedule could then be fed into the launch computers where it would serve as a computerized sequencer for all test procedures. It would be possible to program the computer to operate in two modes: a permission mode wherein it would request permission from each test engineer in turn before proceeding with a test and an exception mode wherein it would proceed with each test automatically until instructed to hold.

The permission mode would be operationally equivalent to the current countdown concept except that the test engineers would play a more passive role, and the launch computer would play a more active role[5].

It should be clear that either of these two modes could incorporate all of the capabilities and flexibility of the semiautomatic scheme such as holds, recycling, etc. Furthermore, it would be possible to resequence tests by simply editing the

5. There have been recent procedural changes that will allow certain no go decisions to be made by computers to allow reduction in countdown staffing requirements that tend to support these concepts.

computer-stored launch schedule either prior to or during a countdown.

One point that was seemingly ignored above in the case of the exception mode is the validation of test results. Many tests are performed for the simple purpose of verifying that a quantum of sampled data is within some previously established set of limits. These limits could easily be encoded along with the other entries in the handbook. In cases where test engineers are essential to validation, the launch computers would be programmed to support man-machine communications.

This discussion has led to a point where we are able by induction to extrapolate the sequencing under computer control beyond the launch countdown and into the mission itself, with the sequencing operation responsibility shifting at some point prior to liftoff to the onboard computer. If we permit arbitrary events, in addition to test and checkout, to be encoded as a function of mission time, then we have a fully automatic, mission-schedule driven, astronaut-in-the-loop system for spacecraft control. This seems like a giant step in automation of space missions, but it is clearly feasible and will no doubt be the standard sooner or later.

What does the mission schedule look like in stored form, and is it necessary for the entire mission to be encoded in advance? The answer to the second question is no, since it would be possible to transmit to the spacecraft a section of the schedule at a time. One minute of transmission time would probably send a schedule that would drive the mission for over 400 days with a mission event rate of one event per minute. It may even be desirable to update the schedule each minute or hour, or the computer could be driven in real time by an on-line ground-based system.

The various parameters required to encode a mission schedule event would include:

- Schedule (mission) time
- Event destination (system monitor)
- Whether a unique control task is required and a task identifier
- An event identifier or code
- Event priority, etc.

Since events are problem oriented (e.g., checkout, experiment, etc.), no general form can be given other than one that would include the five specific items given above in a rigid format (Fig. 6) to enable a program to scan the schedule to recognize events.

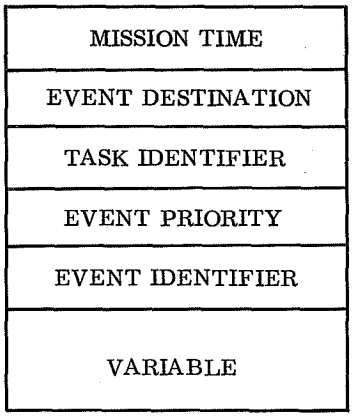


Figure 6. Event entry in mission schedule.

Refer to Figure 3 for a gross flow chart of how the mission schedule influences the system as a result of a scanning operation performed by the executive monitor. Notice that entries from the schedule are queued for distribution to the appropriate destination. It is important to note also that an event is treated as if it were data. Actually, an event is considered as a subset of the class "data" which we have already discussed. The event scheduler shown in Figure 3 has the additional function of making the event conform to the standard data buffer format.

It is not sufficient to treat the queues as FIFO stacks for several reasons, the most obvious of which is the possibility of degradation arising from a relative timing pecularity between events and tasks generated internally by monitors other than the executive, and those of mission schedule origin. Normally, the internally generated information would have higher priority, but it would be necessary to allow the schedule to override in certain cases. A priority operator, referred to as a conflict resolver, has been included to rearrange queues in priority order.

In summary, the mission schedule can be considered as consisting of at least the following functional timelines which are interrelated to form the complete mission:

- Automatic Checkout Timeline
- Astronaut Timeline
- Orbit/Trajectory (Maneuver) Timeline
- Experiments Timeline
- Housekeeping Timeline

These various timelines can drive the complete system through a closely coupled alliance of computer system mission interpretation and astronaut and ground control intervention.

CONCLUSION

A gross-level overview for a supervisor organization has been presented in conceptual form. The features satisfy requirements for mission independence, ease of development, and software reliability. Although this review has been necessarily brief, it exhibits one design approach that is flexible and has many features that allow for comparison of several implementation schemes.

REFERENCES

1. Volpi, R. L.: Apollo Guidance Computers. Electronic Progress, vol. 9, no. 2, 1965, pp. 14-20.
2. Thomas, B. K., Jr.: Apollo 8 Proves Value of Onboard Control. Aviation Week and Space Technology, January 20, 1969.
3. Gruman, E. L.; and Schaenman, P. S.: Functional Requirements of Spaceborne Computers on Advanced Manned Missions. Paper presented at the Spaceborne Multiprocessing Seminar, Museum of Science, Boston, Massachusetts, N68-15439, 1966.
4. Meginnity, D. L.: Advanced Hardware Characteristics of Aerospace Computers. Presentation given at the Spaceborne Computer Software Workshop sponsored by AFSSD and Aerospace Corporation, September 20-22, 1966.
5. Koczela, L.; and Burnett, G.: Advanced Space Missions and Computer Systems. IEEE Transactions of Aerospace and Electronic Systems, vol. AES-4, May 1968, pp. 456-467.
6. Joseph, E. C.: Computers: Trends Toward the Future. Univac.
7. Failure Tolerant Modular Flight Computers. Hamilton Standard System Center, Division of United Aircraft Corporation, Slides SP 02U69, vol. III, January 20, 1969.
8. Andrews, L. J.: Aero/Space Software in Perspective. Proceedings, First Spaceborne Computer Software Workshop, El Segundo, California, September 20-22, 1966.
9. Hokom, R. A.: Executive Program Control for Spaceborne Multiprocessors. Paper presented at Spaceborne Multiprocessing Seminar, Museum of Science, Boston, Massachusetts, N68-15441, 1966.
10. Wirth, N.: On Multiprogramming, Machine Coding, and Computer Organization. Communications of the ACM, vol. 12, no. 9, September 1969.
11. Black, N. E.: An Examination of Spacecraft Computer Requirements for Experiment Data Processing. Computer Sciences Corporation Report, NAS8-18405, September 1968.
12. Quann, J. J.; and Keipert, F. A.: A New Approach to Telemetry Data Processing: The Data Reduction Laboratory. AIAA Aerospace Computer Systems Conference, no. 69-972, September 1969.

REFERENCES (Concluded)

13. Automated Control and Data Acquisition. IBM Journal of Research and Development, vol. 13, no. 1, January 1969.

14. Judge, J. F.: An Airborne Data Explosion. American Aviation, September 16, 1968, p. 66.

15. Dabul, A.: Information Transfer Systems in Space Communications. MSFC Technical Note, NASA TN D-3405.

16. Rosenfeld, A.: Picture Processing by Computer. Academic Press, Inc., 1969.

17. Hewlett-Packard Model 5450 Fourier Analyser. Hewlett-Packard Company, 1501 Page Mill Road, Palo Alto, California 94304.

18. Kerner, H.; and Gellman, L.: Memory Reduction Through Higher Level Language Hardware. AIAA Aerospace Computer Systems Conference. Los Angeles, California, no. 69-963, September 1969.

19. Brewer, M. A.: Adaptive Computers. Information and Control, vol. 11, A68-23159, October 1967, pp. 402-422.

20. ATOLL Reference Manual. NASA, MSFC, R-QUAL-PSC-65-R2.

21. Executive Control Language. UP-4144, Section 5, UNIVAC Data Processing Division, 1968.

Page intentionally left blank

PARALLEL PROCESSING METHODS AND MANNED SPACE MISSIONS

By

M. E. Stegenga

INTRODUCTION

Parallel processing began with the introduction of Solomon in 1959. The use of this concept has continued in such computers as the Univac 1108, CDC 6600, Illiac IV, and modern Fast Fourier Transform computers. Parallel processing concepts and designs were identified to determine which were applicable to a computer for a manned space mission. For example, image processing could be accomplished much faster by having several processors processing portions of an image than by having any one processor processing the image. A parallel processing computer is one capable of simultaneously executing two or more instruction streams.

As an example of parallel processing, consider the computation of D = (A + B) * (E + F). One processor would add A + B while another processor adds E + F. Then, either processor would multiply the two numbers together.

A major reason for using a parallel computer system is to decrease the execution time required for a particular program. This is accomplished by having several processors simultaneously executing portions of a program. Parallel processors are cheaper for several reasons. First, many identical modules are manufactured reducing the development cost; second, the ratio of hardware to throughput is decreased, because, in the case of an array processor, the control hardware is required only once. The exponential nature of the cost versus speed curve for hardware costs means that using several cheaper components to achieve the same throughput of a system can result in a relatively cheaper system. A third aspect of parallel systems is that graceful degradation is possible. This provides a natural means for reliability in that switching out a module will result in less computing capability without causing the system to fail.

The computation reduction ratio is the ratio of execution time on a parallel system to the execution time on a sequential machine with equivalent hardware. An approximate computation reduction ratio for spaceborne applications programs would be 30 [1].

ARRAY PROCESSORS

An array processor increases the speed of a computer by having an array of processors all executing the same instruction simultaneously.

An array processor with N processors all executing an instruction simultaneously will be up to N times as fast without costing N times as much. A system with N computer units (a unit includes a control unit) and N instruction streams would be more versatile and more expensive. (See multiple computer systems.)

Ideally, a compiler should take a sequential program in a source code and transform it into optimized machine code for an array processor. A large number of computation algorithms should be available to enable problems to be computed using many processors.

Illiac IV

The Illiac IV [2-6], a parallel network system, is being developed jointly by the University of Illinois and industry. The computer is to be delivered to the University of Illinois in 1970. The Illiac IV controls a number of data streams with a single instruction stream. It is an array computer consisting of 256 processing elements (PE's), 4 control units, and a B6500 computer. Each processing element has a 2048-word thin-film memory, each word having 64 bits, with a cycle time of 240 nsec. The arithmetic and memory speeds match.

The 256 processing elements are divided into 4 subarrays, each consisting of 64 PE's and having their own control unit. Each subarray is capable of independent processing. The subarrays can be arranged to form 2 subarrays of 128 PE's each or a

single array of 256 processors. The advantages to this arrangement are that failure in any subarray does not preclude continued processing by other subarrays, and the size can be designed to fit a particular problem solution. Figure 1 is a diagram of the Illiac IV system array structure.

The 64 PE's in an array are arranged in a string (Fig. 1) and are controlled by the control unit (CU) that receives the instruction stream and transmits this to PE's for execution. Direct connections exist for nearest neighbors and PE's eight elements away. Thus PE-17 connects to PE-16, PE-18, PE-9, and PE-25. PE-0 connects to PE-63. Local control of a PE is provided by mode control that enables or disables execution of the current instruction.

Data and instructions are stored in the array memories. A CU has access to all memories while a PE has access only to its own memory. There are instructions for transferring data from one PE memory to another along the direct connections.

Basically the Illiac IV is a single instruction stream computer in which each instruction is executed on several processors simultaneously.

Illiac IV Software

INTRODUCTION

Most of the software for Illiac IV [5] (some application programs excepted) is in the design phase. A simple batch processing operating system, an assembler, and a high level language compiler, TRANQUIL, are being designed. Several applications programs have been coded.

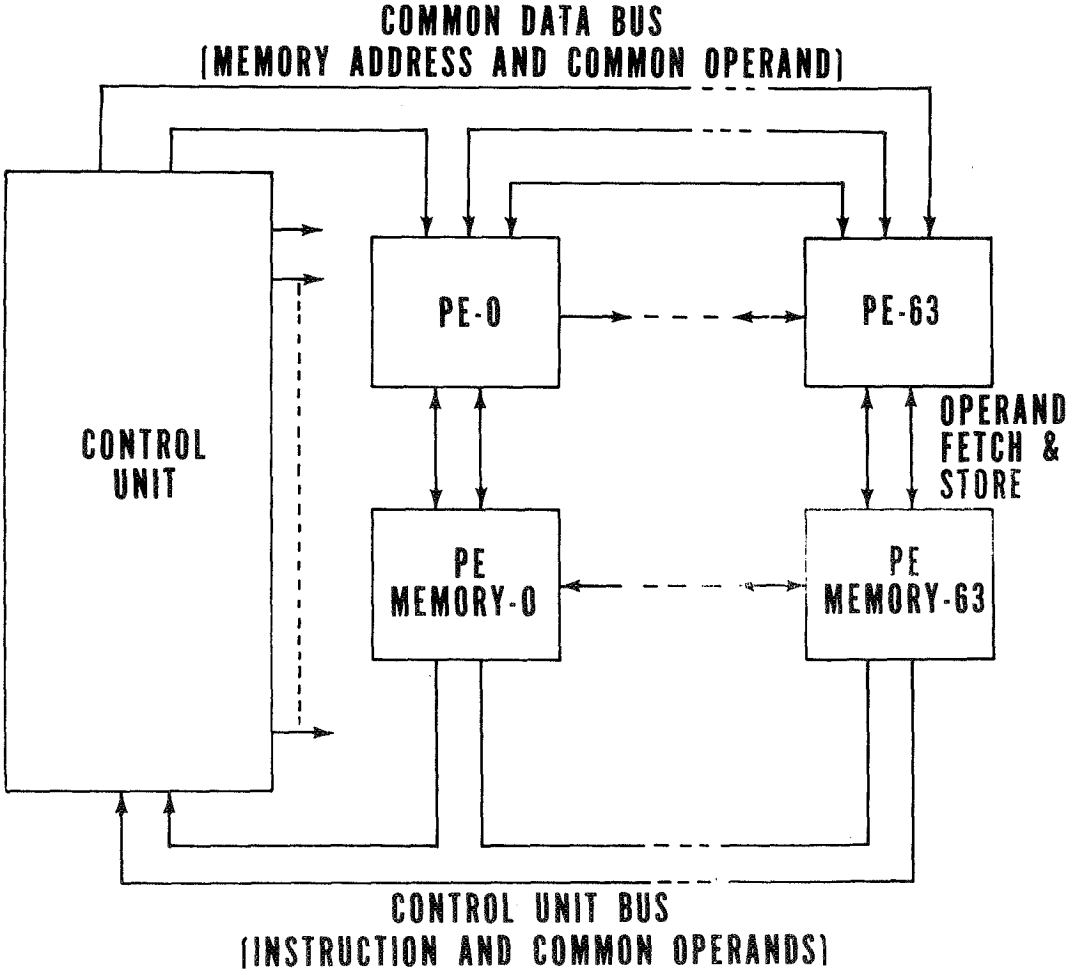


Figure 1. Array structure of the Illiac IV system.

TRANQUIL

TRANQUIL is a higher level language for Illiac IV that will enable users to become familiar with the machine. TRANQUIL is basically ALGOL with some additions and deletions.

An Illiac IV program consists of a storage algorithm and a computation algorithm. TRANQUIL will have variable and constant declarations. The declaration will include word type along with word length. Arrays, in addition, will be declared by the type of storage desired; i.e., straight or skewed. Statements will be included so that an array may be partitioned to best fit the machine. TRANQUIL will have statements to provide for arithmetic operations on either whole arrays or a subarray (a column or row, etc.).

There will be two types of FOR statements, and each type will allow three types of indexing. A single index variable can vary over a single index set. Also, index variables can be paired off with index sets, and the several index variables can be varied simultaneously, over their respective index sets. Finally, a vector can be varied over the cartesian product of index sets. A sequential FOR statement and a simultaneous FOR statement (several instructions executed simultaneously) will be used. Conditional branching statements will be provided so that vectors or subvectors may be compared.

THE HOLLAND MACHINE

General

The Holland machine [7, 8] was originally proposed by John Holland in 1959. He says, "The present formulation is intended as an abstract prototype which, if current component research is successful, could lead to a practical computer." To date, no machine of the Holland type has been built.

Because fabrication costs for large scale integration (LSI) may be small compared to development cost, computer organizations with a large number of identical modules will be more desirable. The Holland machines are examples of such computers. Also, a Holland machine has local control.

The computer is composed of general purpose modules in a two-dimensional rectangular grid (Fig. 2). Each module consists of a binary storage register with associated circuitry and some auxiliary registers. Each module may be called to execute its word as an instruction, to use the register as an accumulator, or to provide the word as data. Since the memory is nonstandard, new concepts are required to replace program execution and data accessing. Each module is connected to its four nearest neighbors; i.e., the four in the vertical and horizontal directions. Each instruction of a module is located in a single module. If a module M_1 executes an instruction, then the successor module, adjacent to M_1 and determined by the bits s_1, s_2 in the auxiliary register of M_1, is executed unless the instruction is the equivalent to a transfer instruction. The predecessor module is determined by the bits q_1, q_2 in the auxiliary register. Since more than one module can be active at one time, several subprograms can be executed simultaneously.

The action of a module during a time step can be divided into three successive phases.

1. Input Phase: A module's storage register can be set to any number supplied by a source external to the computer.

2. Path Building Phase: An active module determines the location of the operand; i.e., the storage register upon which the instruction is to operate.

3. Execution Phase: The active module interprets and executes the operation in its storage register.

Input

During the initial phase of each time step, a module's storage register can be set to a chosen value. It is expected that most of the modules will receive input only when storing the program.

Path Building

If bit P in the auxiliary register is set to 1, then the module is designated as a P-module. A path is divided into straight-line segments. An

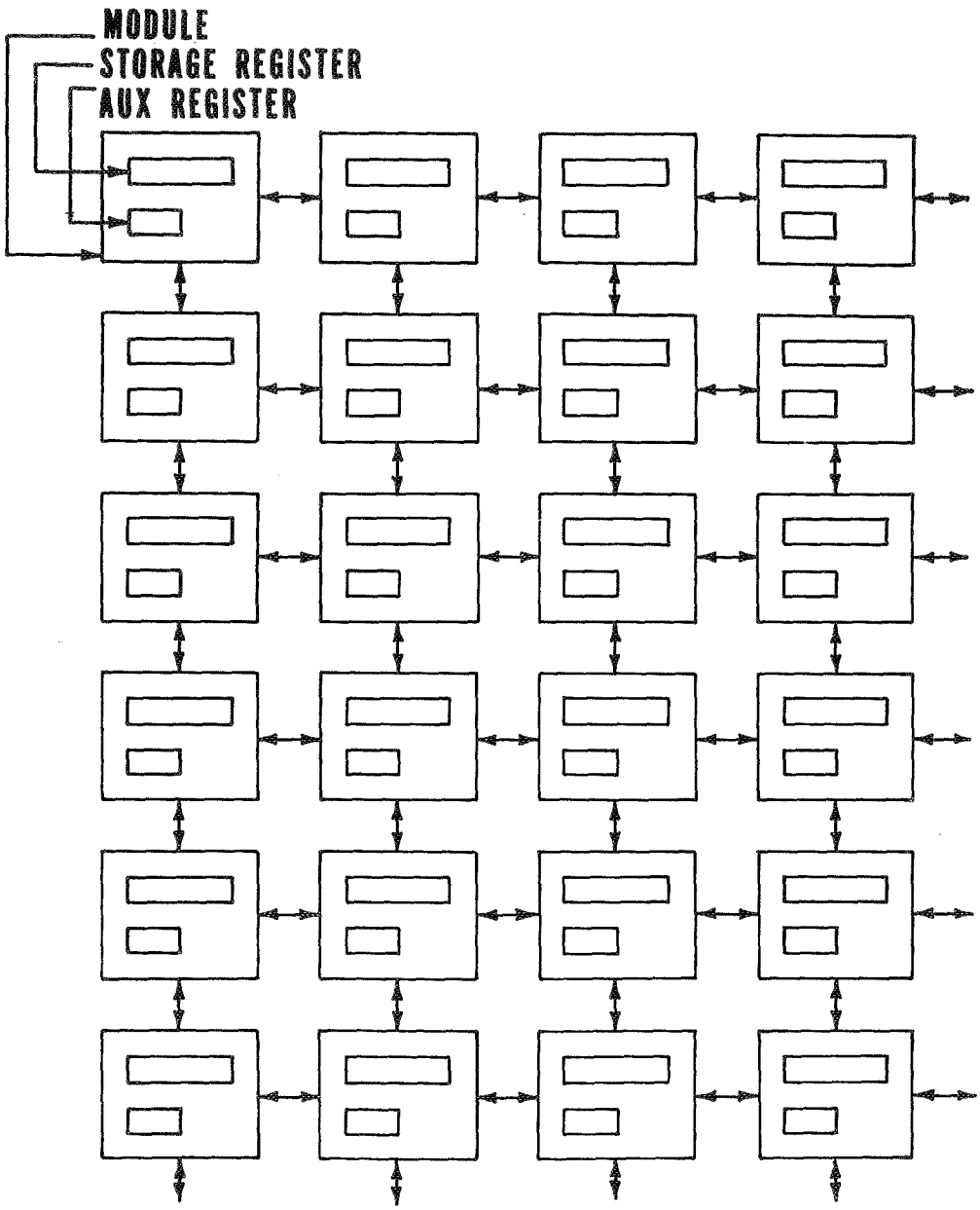


Figure 2. Array structure of the Holland Machine.

instruction can, at most, add or delete one segment from a path or begin a path with a segment. All paths begin at P-modules.

Building a path consists of the following three steps: (1) locate the starting point; (2) if a path already exists there, then move to its termination point; and (3) build the segment in the direction and length specified.

First, the path specification bits $y_o, \ldots, y_n$ and d_1, d_2 in the auxiliary register are gated over the line of predecessors to the first P-module. If a path begins at a module, then the appropriate star register, $(b_1 b_2)^*$ in the auxiliary register of a module is turned on. A path once marked persists until erased. The path determination data is then gated to the end of the path, and a new path is

constructed of length $\sum_{k=0}^{n-1} y_k$ and direction (d_1, d_2) provided $y_n = 0$. If $y_n \neq 0$, then the last segment is erased.

Execution

The active module contains the operation code in bits $i_1, \ldots, i_4$. The path termination of the first preceding P-module contains the operand in its storage register. There must be a module that serves as an accumulator. A module is an A-module if, and only if, the bits (P,A) are set to (0,1). Thus, no P-module can be an A-module. The accumulator is the first A-module encountered along the line of predecessors.

The execution of an instruction occurs in the following sequence:

1. The operation code is sent down the line of predecessors to the nearest A-module.

2. When the operation code passes through the P-module, a signal is sent down the path from the P-module to its termination indicating that this is a fetch command.

3. The operand (data) is sent to the appropriate A-module via the P-module.

4. The operation is accomplished. The operation could be performed as the operand arrives at the A-module.

5. A completion pulse is sent to the active module via the line of successors.

6. The active module turns itself off and its successor on.

Summary

With the present state-of-the-art technology, such a computer would be expensive to build because of the large amount of hardware involved. With the advent of LSI, such a computer seems feasible. A related disadvantage is the low rate of hardware utilization. Another important problem is the difficulty in programming such a machine, although several people think that a compiler is within the state-of-the-art.

A DISTRIBUTED PROCESSOR

General

The distributed processor machine was designed as an onboard computer that would take full advantage of LSI technology projected for the late 1970's. It is a parallel processing machine designed so that graceful degradation is possible. In one state it can function as an array processor and in another state it can function as a distributed processor (the computations are distributed over the modules). Programming such a computer would be quite difficult.

The distributed processor is divided into groups where each group consists of a collection of cells or modules [1,9] (Fig. 3). The cells in a group are interconnected (neighboring cells are connected and there is an intercell bus) and groups are connected by an intergroup bus. Each cell has its own 16-bit word, 512-word memory. In addition, there is a bulk memory unit for loading and unloading the cells. Parallelism can exist between cells of a group and between groups. Also, a group or part of a group can function as an array processor. In Reference 1 the system is organized into four groups of 20 cells each. There is an executive and an executive backup for the system of four groups, and each group has its own executive in a controller cell.

Cells

All cells in a group are identical. Each cell is provided with several accumulators and at least three index registers to reduce memory requirements. The word length is 16 bits. Complete arithmetic and control functions are incorporated in each cell. A cell is provided with several possible states so that both local and global control can be utilized. The possible cell states are listed below:

1. Permanently failed — power off
2. Shutdown — power saving state
3. Independent
4. Dependent under global control (global state)
5. Dependent under local control

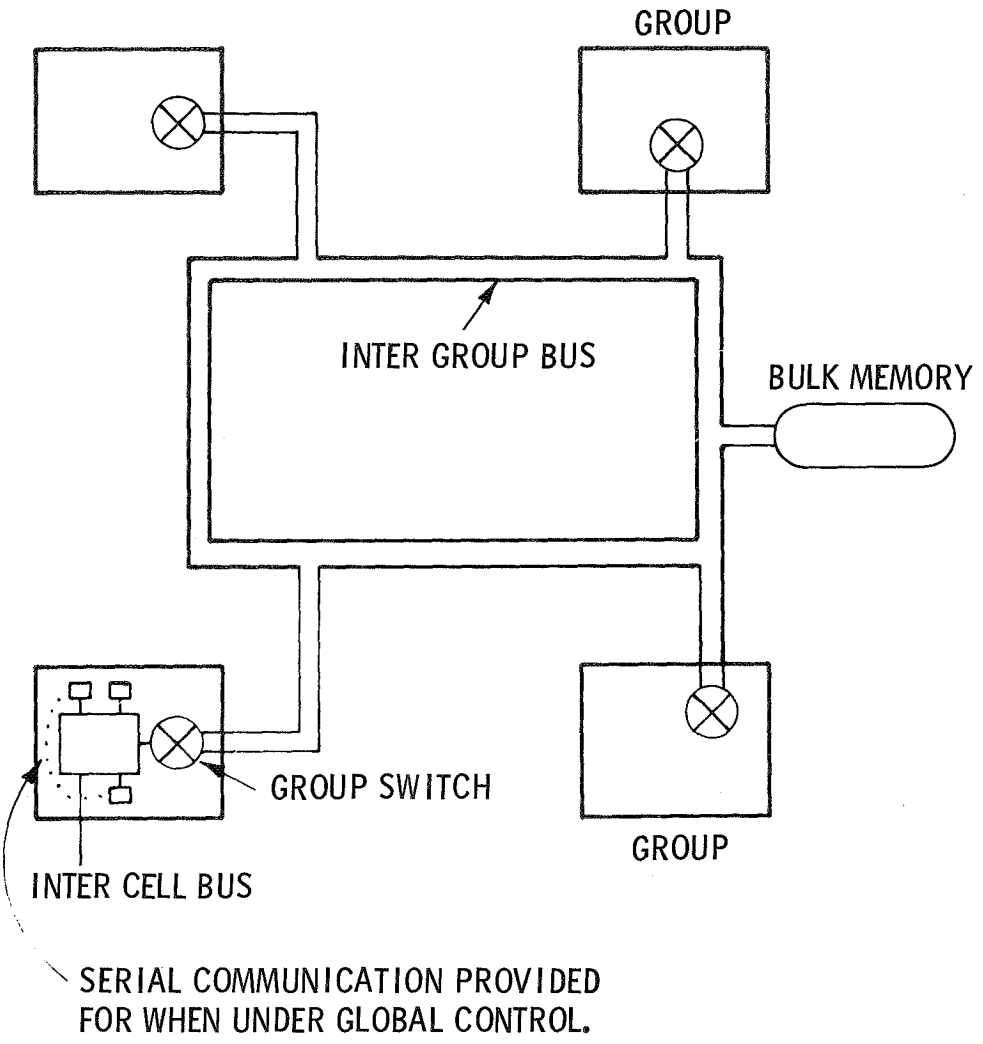


Figure 3. Distributed processor system.

6. Dependent in wait state

7. Controller cell

Independent cells function essentially the same as a sequential computer. They fetch instructions and operands from their own memory and execute them. A cell remains in this state until the controller cell commands it to change states. Dependent cells are under the control of the controller cell, the cell which controls the cells of a group.

Each cell is designed to fit on an LSI chip. Approximately one-tenth of a cell's hardware is in the processor portion and nine-tenths are in the memory section. A cell, if necessary, can use memory space in adjacent cells. Each cell can function in any one of the seven states.

Software

Programming such a computer will be quite complex, and no work has been done on a compiler as yet. The executive consists of three parts — the system executive, the group executive, and the cell executive.

A MULTIPLE COMPUTER SYSTEM

General

A multiple computer system decreases the execution time of a program by having many modules simultaneously computing different portions of a program. It is a general purpose system with many instruction streams executing simultaneously. The primary advantages to this type of system are that a program can be executed faster and that the system can be gracefully degraded.

A multiple computer system is a computer consisting of several (more than 1) processors and several memory units which, in addition, is capable of simultaneously executing several different instruction streams. A multiple computer system is a more general purpose system than an array processor.

Curtin's Computer

Curtin's computer system [10] consists primarily of M memory units nominally of 4096 words storage each, and N computers where the logic and arithmetic functions are combined. The communication between computer and memory is provided by a data bus. Each computer has access to memory once each memory cycle. This concept is a time division of the communication medium. Each computer would have several memories (at least one) assigned to it. Communication between computers is accomplished by reassigning memories from one computer to another. A separate data transfer bus would be provided between the input/output and the memories. This system makes parallel computations possible in that different computers can compute different portions of a program. Programming this machine would be difficult, because each separate computer must be programmed individually. No facilities are given for timing separate sequences of a program. For a general diagram of Curtin's computer, see Figure 4.

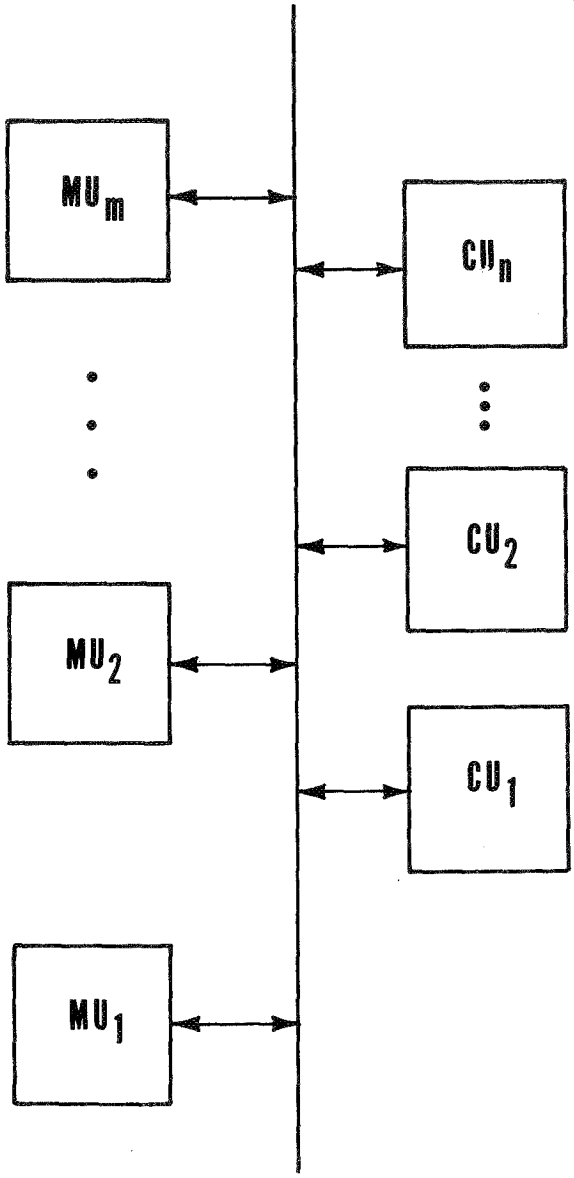


Figure 4. Multiple computer system.

CONCLUSIONS

Illiac IV is a problem-oriented machine and is most applicable for solving such problems as convolutions, Fourier transforms, partial differential equations, matrix manipulations, and weather predictions. It is an example of an array processor. Illiac IV has a distributed memory, a method for broadcasting data to all processing elements, and a technique for determining whether or not a particular processor executes an instruction. The latter is useful for branching.

An array processor would be very inefficient as a general purpose computer, because many modules would be inactive at any particular instant of time. Some problems, such as orbit prediction, are not amenable to solution on an array processor with many processing elements. Such a problem could

be subdivided into different tasks to be solved on separate modules. This would require several instruction streams. Thus, the overall concept of an onboard computer should not be that of an array processor. An array processor could, however, be used as a module in a multiple computer configuration. Then, certain types of problems, such as transformation of coordinates, would be solved more quickly and efficiently at less cost per computation. A major advantage of an array processor is the speed of execution. A significant disadvantage is that, except for special applications, the array processor is inefficiently used.

The multiple computer system seems to be an excellent architecture for use as an onboard computer system. Several program streams can be executed simultaneously (as many instruction streams as there are modules). Graceful degradation can be accomplished by switching out a module as soon as it fails. Programs would be executed, under priority, only when modules become available to run them. Parallel processing could be applied when necessary to speed up the execution of a program. The parallelism of a program could be specified by the use of Fork and Join statements; or, if the state of the art changes, a compiler could be used to transform a sequential program. Special modules could be incorporated, such as an array processor, for particular applications.

The distributed processor combines features of an array processor, a Holland machine, and a multiple computer system. The ability of modules (cells) in a group to function as an array processor or as individual elements in a multiple-computer-type architecture is one feature that could be applied to any multiple computer system. Incorporating this feature reduces memory requirements and simplifies some application programming in this system. The distributed processor allows for graceful degradation and the switching-in of replacement modules. Because of its unusual architecture, considerable design work would be necessary. The time frame (1978) of its design appears to be too distant for immediate use on a space station. For interplanetary distances and reliability periods of several years (it was designed for this purpose), its architecture would be applicable. Programming the distributed processor would be difficult, and a compiler is beyond the present state-of-the-art. Some functional designs of an executive have been done.

The Holland machine is not developed enough for 1975 and provides poor utilization of available hardware. Local control, modularization, and graceful degradation are particularly interesting for possible incorporation into a spaceborne computer system. The computer architecture which seems most applicable as an onboard computer system is that of a multiple computer system.

Below is a table presenting the subjective evaluation of the computer systems in this report in tabular form. A value of 1 means that the system does not have the property, a value of 2 means the system possesses the property to some degree, and a value of 3 means the system possesses the property to a large degree. The basis of comparison is the CDC 6600. "H" will represent a Holland-type machine, "A" an array processor, "MC" a multiple computer system, and "DP" a distributed processor.

Characteristic	DP	MC	A	6600	H
Graceful Degradation	3	3	1	2	3
Speed	3	3	3	2	1
Programmability	1	2	2	3	1
Low Cost per Computation	2	2	3	2	1
Versatility	3	3	1	3	2
Within State-of-the-Art	1	2	3	3	1

REFERENCES

1. Koczela, L. J.: The Distributed Processor Organization. Advances in Computers, Academic Press, New York, 1968, pp. 286-353.

2. Barnes, G. H., et al.: The ILLIAC IV Computer. IEEE Transactions on Computers, vol. C-17, August 1968, pp. 746-758.

3. ILLIAC IV Quarterly Progress Report. July, August, September, AD665916, 1967.

4. ILLIAC IV Quarterly Progress Report. October, November, December, AD667280, 1967.

5. Kuck, D. J.: ILLIAC IV Software and Application Programming. IEEE Transactions on Computers, vol. C-17, August 1968, pp. 758-770.

6. Stokes, R. A.: ILLIAC IV: Route to Parallel Computers. Electronic Design, 26, December 20, 1967, pp. 64-69.

7. Holland, J. H.: A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously. Proc. E.J.C.C., 1959, pp. 108-113.

8. Holland, J. H.: Iterative Circuit Computers. Proc. W.J.C.C., 1962, pp. 259-265.

9. Burnett, G. J.; Koczela, L. J.; and Hokom, R. A.: A Distributed Processing System for General Purpose Computing. Proc. F.J.C.C., 1967, pp. 757-768.

10. Curtin, W. A.: Multiple Computer Systems. Advances in Computers, vol. 4, 1963, pp. 245-301.

Page intentionally left blank

OPTIMIZATION OF AN INSTRUCTION SET FOR A GENERAL PURPOSE SPACEBORNE COMPUTER

By

J. R. Kennedy

ABSTRACT

The development of a general purpose digital computer is at best a compromise between macroinstruction flexibility versus hardware logic complexity on one hand and application data word sizes and data path width on the other. When the machine is designed with stored logic control, the first tradeoff burden is eased considerably to the extent that macroinstruction flexibility is compared with the cost of read-only-memory (ROM). The data path width should be considered simultaneously with the organization of the macroinstruction set to determine an optimal data path width including memory bus and register widths. The final word size should therefore be based on data precision and instruction bit requirements. With these ideas in mind, the problem was envisioned and structured throughout the development of this report as follows:

1. Given a set of functional registers and a feasible, conjectured instruction format, develop a basic instruction set to perform the general purpose functions of control including:

   a. Register load and store

   b. Register content logical manipulation

   c. Register content arithmetic

   d. Instruction sequence routing

   e. Procedure linkage

   f. General control

2. Analyze the instruction set developed in step 1 for similarities among various instructions and interrelations among the various fields; as a result of the analysis, take advantage of various tradeoffs and relations to enable a more efficient organization of the registers, fields, and instruction formats with the objective of improving memory utilization (efficiency) both in main core memory and the ROM.

3. Consider data precision requirements, and adjust the main memory word size (bus width) to optimize manipulation of both data and instructions.

4. Specify the resulting preferred instruction set and formats, data formats, and affected data routing schemes.

SUMMARY

For the purposes of this report, a microprogrammed digital computer is envisioned as a general purpose processor having a scratch pad memory accessible by macrolevel programs, four accumulators, four base registers, and four index registers along with a potentially useful instruction word formatting scheme.

A set of space-oriented functional requirements is first transformed into a set of computer operations containing advanced and basic operations. Then, with the stated assumptions regarding the computer organization, a basic instruction repertoire is developed.

Next the repertoire is studied to ascertain its weaknesses and their causes. This analysis leads to consideration of alternate organizational schemes having to do primarily with formatting and register usage. Modifications are recommended that are shown to (1) reduce the number of instructions in the repertoire by 17 percent, (2) reduce the number of format configurations by 54 percent, (3) reduce the amount of coding required for a sample program by 28 percent, (4) give greater programming flexibility, and (5) make it possible to expand the repertoire by a larger factor as requirements grow.

This discussion is a condensation of a more detailed report titled "Basic Instruction Set for a

Proposed 24 Bit General Purpose Spaceborne Digital Computer," dated 13 August 1969.

INTRODUCTION

This section gives a brief description of a 24-bit digital computer that is controlled by the microprogrammed ROM. A block diagram is included along with a descriptive trace of an instruction word fetched from memory.

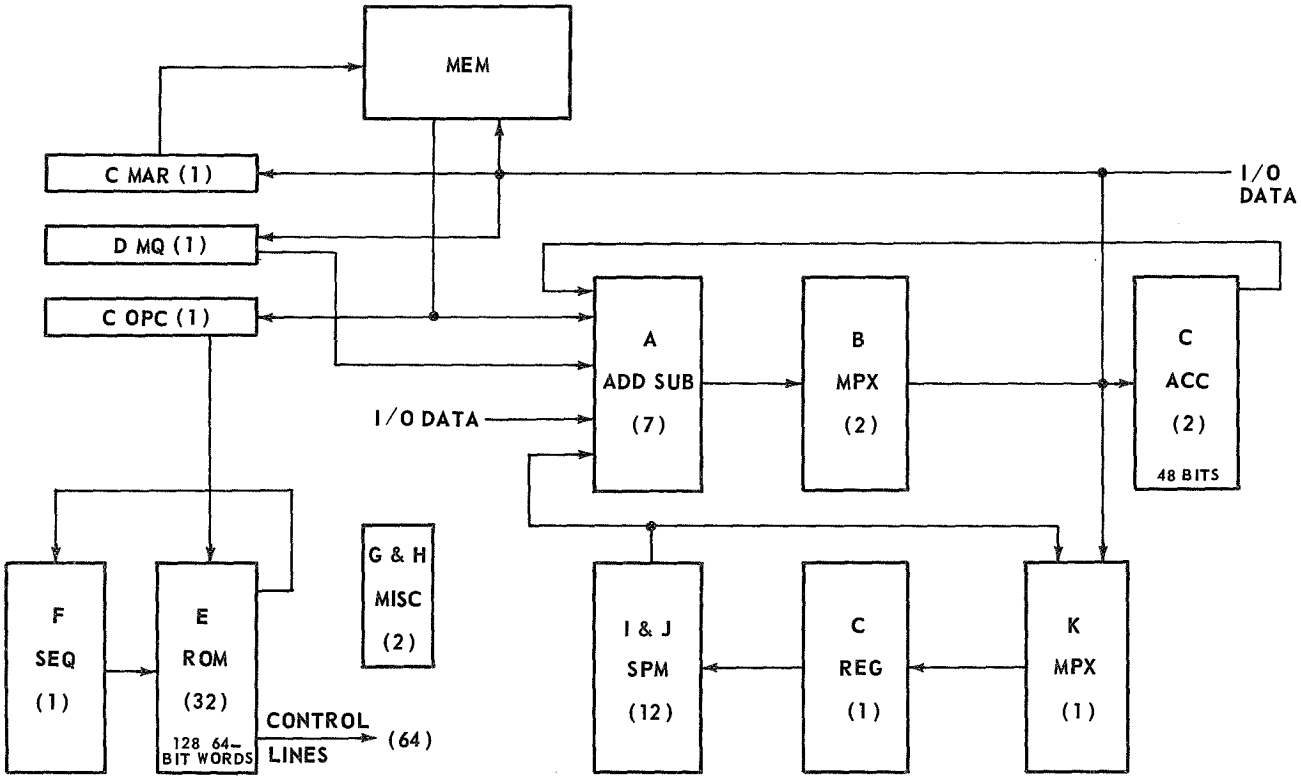
Block Diagram

Figure 1 shows a gross-level block diagram for the specified 24-bit word length computer. The normal sequence of operation is as follows:

1. A word (program address counter) in the ROM is accessed for the main core memory address of the next instruction.

2. This address is loaded into the MAR, and main core memory is strobed to read the contents into the instruction register (OPC). The operation code field is translated through an ROM address map into the sequencer for the initial ROM microword location and all subsequent microwords required to interpret this instruction. The microcode sequence will update the program address counter for the next instruction fetch cycle.

The above is not concerned with details of instruction decoding or data word fetches from main core memory; the intent is only to give an intuitive feel.



BLOCK DIAGRAM NOTATION

ADD/SUB	4-Bit Adder Register	OPC	24-Bit Op Code (Instruction) Register
ACC	48-Bit Accumulator Register	SEQ	8-Bit Sequencer
SPM	24-Bit Scratch Pad Memory	ROM	64-Bit Read Only Memory
MAR	24-Bit Memory Address Register	MEM	24-Bit Main Core Memory
MQ	24-Bit Multiplier/Quotient Register	MISC	Miscellaneous

Figure 1. Computer block diagram.

Cost Factors

The development of a general purpose digital computer is at best a compromise between macroinstruction flexibility versus hardware logic complexity on one hand and application data word sizes and data path width on the other. When the machine is designed with stored logic control, the first tradeoff burden is eased considerably to the extent that macroinstruction flexibility is compared with the cost of ROM. The data path width should be considered simultaneously with the organization of the macroinstruction set to determine an optimal data path width including memory bus and register widths. The final word size should therefore be based on data precision and instruction bit requirements. In most cases various tradeoff analyses result in a bus and register width that is smaller than required by a majority of the data precision and instruction formats. At this point, a decision is usually made as follows:

1. If the computer is to be used for general purpose scientific problem solution where high throughput is a necessity, the bus and register width tends toward favoring high data precision and long instruction formats. In this case the high throughput is "paid for" by emaciation of useful memory because of unused bits, particularly where instructions are concerned.

2. If the machine is to be used in a dedicated way and flexibility is required along with relatively low cost, then memory conservation and logic simplicity are the guidelines, and the bus width invariably shrinks to the lowest viable width consistent with reasonable throughput capability.

With these ideas in mind the task was envisioned and structured throughout the development of this report as the following problem:

1. Given a set of functional registers and a feasible, conjectured instruction format, develop a basic instruction set to perform the general purpose functions of control including:

   a. Register load and store

   b. Register content logical manipulation

   c. Register content arithmetic

   d. Instruction sequence routing

   e. Procedure linkage

   f. General control

2. Analyze the instruction set developed in step 1 for similarities among various instructions and interrelations among the various fields; as a result of the analysis, take advantage of various tradeoffs and relations to enable a more efficient organization of the registers, fields, and instruction formats with the objectives of improving memory utilization (efficiency) both in main core memory and the ROM.

3. Consider data precision requirements, and adjust the main memory word size (bus width) to optimize manipulation of both data and instructions.

4. Specify the resulting preferred instruction set and formats, data formats, and affected data routing schemes.

The following topic specifies the basic instruction format and summarizes the instruction set; the section entitled "Impact Analysis" discusses an alternate approach to register usage and shows the improvements that result. This report represents a condensed version of the original. The original report includes more detailed considerations with examples and a discussion of scratch pad memory usage, interrupts, debug features, etc.

BASIC INSTRUCTION SET

This section outlines a basic instruction set to allow stored program control of the specified general purpose digital computer. The initial instruction set is designed around a given word format to provide for basic logical and arithmetic control functions at the macroinstruction level. An attempt is made to describe the individual instructions to a level of detail that allows for unambiguous comprehension but does not hamper the freedom of development of interpretation logic at the microprogramming level.

Functional Computer Requirements

The subject computer is viewed as a potential candidate for space station digital computations including the following problem oriented functions:

1. Input/Output (I/O)
    a. Analog-to-Digital Converter Sampling (Input)
    b. Digital-to-Analog Converter Control (Output)
    c. Display Beam Driving (Point Coordinates) (Output)
    d. Character Message (Input and Output)
    e. I/O Device Status Sensing (Input)
    f. Device Controller/Multiplexer Function Command (Output)
2. Interpretation
    a. Tests and Comparisons
        (1) Branch on Equal
        (2) Scan Further on Not Equal
        (3) Memory/Register Content Comparison
    b. Procedure Linkage and Return Control
    c. Conditional and Unconditional Instruction Sequencing
3. Process Control for Experiments
    a. Interval Clock
        (1) Set Timer
        (2) Queue Requests
    b. Calculate Control Values
        (1) Fixed Point Arithmetic
    c. Sample Data Automatic
        (1) Scanner Interval
        (2) Storage Management
4. Navigation
    a. Transformations
    b. Floating Point Arithmetic
    c. Vector Algebra
    d. Trigonometry Functions
5. Guidance/Control
    a. Most of the Above

Based on the above problem-oriented requirements, the list below indicates potential candidates for inclusion in a total instruction repertoire.

1. Arithmetic
    a. Floating Point
    b. Fixed Point *
2. Queue/Stack
    a. Pointer Manipulation *
    b. Data Component Packing/ Unpacking *
    c. List Data Processing
3. Test/Compare *
    a. Register with Memory
    b. Inter-Register
4. Vector (Floating Point)
    a. Dot Product
    b. Add
    c. Subtract
5. Trigonometry
    a. Sine
    b. Cosine

c. Square Root

d. Tangent

6. Subroutine Linkage *

a. Call

b. Return

7. Interval Timer Control *

8. Program Address Counter Access *

9. Instruction Sequence Control *

a. Unconditional Branch

b. Conditional Branch

10. Bit Manipulation *

a. Shift

b. Rotate

11. Character Manipulation

12. Interrupt Control *

a. Enable/Disable

b. Status Sense/Clear

13. Input/Output

Of the candidates listed above, those having an asterisk to their right were considered necessary to provide a basic control capability. Those not starred are considered to have undefined properties or were excluded because they are not justifiable without further requirements analysis.

Instruction Formats

Two instruction formats based on a 24-bit main memory word length are specified as shown in Figure 2. The fields are defined as follows:

OP CODE : 6-bit operation code provides for 63 memory reference instructions.

A : 2-bit accumulator provides for specification of 1 of 4 possible accumulators numbered (addressed) 0 to 3.

B : 2-bit base register provides for specification of 1 of 4 possible base registers addressed 0 to 3.

X : 2-bit index register provides for specification of 1 of 4 possible index registers addressed 0 to 3.

DISP : 12-bit displacement provides for specification of a constant K where $0 \leq K \leq 4095$.

EOP : 6-bit extended operation code provides for an additional 64 nonmemory reference instructions.

SPM : 6-bit scratch pad memory address provides for referencing high speed memory location K where $0 \leq K \leq 63$.

Two exceptions that were taken in the development of the instruction set are explained later.

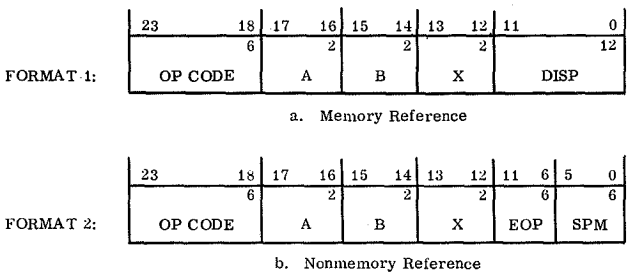


Figure 2. Instruction word formats.

Register Functions

ACCUMULATORS

The four accumulators are used as working registers in that all main core memory data fetches and stores take place via these registers; shift operations, arithmetic, testing, and miscellaneous functions also make use of these utility registers.

BASE REGISTERS

Because of the small (24-bit) word size combined with a possibly large (on the order of 100 000 or more words) main memory, it is not possible to completely specify the absolute address of a datum within an instruction word. Primarily for this reason (there are other procedure organization and data structuring benefits), the effective absolute memory address is formed by making use of one of these 24-bit registers. This allows memory to be expanded to 2^{24} locations depending on storage requirements and weight only.

INDEX REGISTERS

To allow array indexing operations, these registers are treated in a functionally identical fashion to the base registers. In conjunction with the value, d, specified in the displacement field, the content of the specified index register (x) is combined with the content of the specified base register (b) to calculate the effective absolute memory address, E, of the instruction operand. Thus, E = (b) + (x) + d, where enclosure in parentheses implies "contents of" the specified register. For reasons that are explained in detail in "Impact Analysis," index register "0" is not available so that the address "0" can be used to indicate that E is to be calculated as E = (b) + d, with no indexing applied.

Instruction Summary

Table 1 gives a condensed summary of the basic instruction repertoire. A more detailed explanation can be found in the complete report.

IMPACT ANALYSIS

This section summarizes features of the instruction set of Table 1 according to format and compares an alternative approach to providing a functionally equivalent repertoire that may require less memory for implementation in the ROM and less memory for programming in the main memory.

Distribution of Instructions

The instruction repertoire summarized under "Basic Instruction Set" was analyzed and compared with instruction sets for several existing, widely-used computers. It is considered to be basic from the point of view that no sophisticated capabilities are provided, but it is certainly not minimal in that certain operations can be performed by the use of several other operations in sequence.

Of interest now is a table showing the distribution of instructions according to format and field usage. Table 2 shows this in condensed form where the number of instructions in each configuration is shown in column NI. Notice that although only one format is represented in Table 2.a, several different configurations are represented, and only 24 of 50 instructions make use of all the fields specified by the format. In Table 2.b, six different configurations are found and none of the instructions make use of all the fields specified by the format.

These interesting observations show that 52 of the total of 76 instructions (or about 68 percent) do not make use of all the fields provided. Since the space provided for the fields represents wasted main memory when it is not used, we can expect a program written using the scheme specified in the topic entitled "Basic Instruction Set" to make inefficient use of main memory.

In addition to the above considerations, inspection of the instruction set in the topic entitled "Basic Instruction Set" will show that there are several instructions devoted to transferring information from one register to another, as from an accumulator to a base. This could be avoided by having an instruction to load the base from memory and by allowing arithmetic and logical operations to be performed using the base contents as an operand.

Two fundamental weaknesses in the specified approach are concluded as follow:

1. The formats cause inefficient main memory usage because of frequent occurrence of unused fields.

2. The register usage restrictions cause main memory inefficiencies.

The obvious question is, therefore: Given the basic capability exhibited by the specified instruction repertoire, is there a more efficient formatting and register usage scheme? If so, what effect would it have on the programming of ROM; would it be more extensive or more complicated?

TABLE 1. INSTRUCTION SUMMARY

Mnemonic	Arguments	Short
a. Format 1		
LA	a b x d	Load Accumulator
SA	a b x d	Store Accumulator
LD	a b x d	Load Double
SD	a b x d	Store Double
LM	a b x d	Load Masked
SM	a b x d	Store Masked
SP	b x d	Store p in Memory
SRA	a b x d	Store Right Accumulator
CLM	b x d	Clear Memory
OAM	a b x d	Or Accumulator to Memory
AAM	a b x d	And Accumulator to Memory
EAM	a b x d	Exclusive Or Accumulator to Memory
OMA	a b x d	Or Memory to Accumulator
AMA	a b x d	And Memory to Accumulator
EMA	a b x d	Exclusive Or Memory to Accumulator
CM	b x d	Complement Memory
ADD	a b x d	Add Memory to Accumulator
SBR	a b x d	Subtract Memory from Accumulator
MLT	a b x d	Multiply Accumulator by Memory
DVD	a b x d	Divide Accumulator by Memory
SMZ	b x d	Skip on Memory Zero
SMN	b x d	Skip on Memory Not Zero
SMP	b x d	Skip on Memory Plus
SMM	b x d	Skip on Memory Minus
SALM	a b x d	Skip on Accumulator less than Memory
SAEM	a b x d	Skip on Accumulator Equal to Memory
SANM	a b x d	Skip on Accumulator Not Equal to Memory
SAGM	a b x d	Skip on Accumulator Greater than Memory
EXC	a b x d	Execute
JP	b x d	Jump
ALI	a d	Accumulator Less than Immediate
AEI	a d	Accumulator Equal to Immediate
ANI	a d	Accumulator Not Equal to Immediate
AGI	a d	Accumulator Greater than Immediate
LEA	a b x d	Load Effective Address
SAL	a b x d	Set Alarm
IIS	a x d	Increment Index
DIS	a x d	Decrement Index
SSA	d	Set Stall Alarm

TABLE 1. (Concluded)

Mnemonic	Arguments	Short
IA	a d	Increment Accumulator
IB	b d	Increment Base
IX	x d	Increment Index
LKA	a d	Load Constant into Accumulator
LKB	b d	Load Constant into Base
LKX	x d	Load Constant into Index
R	a x d	Rotate
RP	a x d	Rotate Pair
S	a x d	Shift
SP	a x d	Shift Pair
LJP	b x d	Load and Jump
b. Format 2		
HPR		Halt and Proceed
NOP		No Operation
LJA	a	Last Jump Address
SPA	a	Store p in Accumulator
SIS	a	Sense Interrupt Status
SIM	a	Set Interrupt Mask
RIP	a	Return from Interrupt Process
DSB		Disable Interrupts
ENB		Enable Interrupts
CIM	a	Clear Interrupt Mask
SXP	x	Skip on Index Plus
SXM	x	Skip on Index Minus
SXZ	x	Skip on Index Zero
SXN	x	Skip on Index Not Zero
SALX	a x	Skip on Accumulator Less than Index
SAEX	a x	Skip on Accumulator Equal to Index
SANX	a x	Skip on Accumulator Not Equal to Index
SAGX	a x	Skip on Accumulator Greater than Index
RSA		Reset Stall Alarm
CA	a	Complement Accumulator
CB	b	Complement Base
CX	x	Complement Index
TAB	a b	Transfer Accumulator to Base
TBA	a b	Transfer Base to Accumulator
TAX	a x	Transfer Accumulator to Index
TXA	a x	Transfer Index to Accumulator

TABLE 2. INSTRUCTION DISTRIBUTION

a. Format 1 Configurations

OP	A	B	X	D	NI
x	x	x	x	x	24
x		x	x	x	9
x			x	x	2
x				x	1
x	x		x	x	6
x	x			x	6
x		x		x	2
			Total		50

b. Format 2 Configurations

OP	A	B	X	S	XOP	NI
x					x	5
x	x				x	7
x		x			x	1
x			x		x	5
x	x	x			x	2
x	x		x		x	6
				Total		26

Register Usage

REGISTER FIELD SPECIFICATION

Inspection of the formats depicted in Table 2 shows that a predominant feature existing in nearly all the configurations is that at least one register is specified. In some cases, only one register is required.

Also, it should be noted that, by definition, base and index registers are functionally identical. That is, they are both intended to be used in the same (additive) way in calculating effective main memory addresses. Note, however, that in order to implement certain instructions within the framework of the specified formats, these registers have been used for other purposes (consider IIS, DIS, IA, etc.). This seems to indicate that the registers are not functionally dedicated to the task of indexing.

It is clear that rather than add more instructions to allow base and index registers to be manipulated, there should be a search for a scheme to specify which type and which unique register is desired — all within a single field that can be used to identify any register; and, in cases where the type specification is critical, it must be specified within the operation code field. This scheme would provide a field that can be designated as "R" (register) with no particular function associated with the field other than a register addressing function.

REGISTER ADDRESSING

A way to address the various registers and to derive field formats for inclusion in an instruction needs to be specified. Notice that two aspects of register usage are of concern: one is how a given register is to be treated functionally during instruction execution and the other is which registers will be allowed to be addressed within given fields.

The formats shown in Figure 2 allow for two bits each to address four accumulators, four base registers, and four index registers, all dedicated. It was decided that only three of the index registers would be defined so that the fourth address could be used to specify that no indexing was to be applied in calculating effective addresses. This choice was felt to be better than forcing the programmer to clear an index to prevent incorrect address calculation (in practice, this would result in only three effective index registers). The result of this dedicated register scheme, therefore, is that 6 bits are required to address 11 (or 12) registers; and even though the address of a base register in the A field can be specified, the microprogram will always take this address to be that of the dedicated accumulator with the same address.

If these same 12 registers, whose properties are identical, are specified within an instruction format that specifies one functional R-field and two functional X-fields (for indexing) and all registers are allowed to be addressed in all fields, 12 bits (4 for each field) are needed. Since 4 bits allow up to 16 registers to be addressed in each field, all 16 registers may as well be defined to allow greater flexibility. Clearly, there are many addressing schemes; and in order to analyze various ways to specify formatting, we establish the following features:

1. A register function is specified by the instruction field in which it appears. Field functions are defined as follows:

   a. R-field: a register-addressing field wherein the associated function is specified only by the operation code field.

b. I-field: a register addressing field dedicated to the specification of an indexing function; microcode always associates an indexing function with registers addressed by this field. (If there are two I-fields, they will be referred to as B and X.)

2. Registers are grouped by type as follows:

a. A-type: Registers of this type belong to a group whose members can have their contents used in all manipulative operations exclusive of the calculation of effective main memory addresses.

b. B-type: Registers with this type designation belong to a group whose members can have their contents used in the calculation of effective main memory addresses.

c. X-type: Registers with this type designation belong to a second group whose members can have their contents used in the calculation of effective main memory addresses.

d. G-type: Registers whose type designation is A, B, and X.

Figure 3 shows several schemes for register usage that can be considered for possible implementation. Scheme 1 consists solely of general (G-type) registers. From the macroprogramming and microprogramming points of view, this is the best scheme. However, reference to Table 3 will show by comparison that for a given number of registers, it costs the most in terms of total bits for field addressing.

Scheme 2 is the least expensive in terms of required addressing bits but is the least desirable from the programming and efficiency point of view. (Results in one type register are continually needed in a context (field) in which they can not be referenced.)

Scheme 3 is a hybrid approach, as is scheme 4. Scheme 4 is the nearest scheme to an even tradeoff in terms of programmability and cost. This can be seen in the summary (Table 4) and schematically in Figure 4.

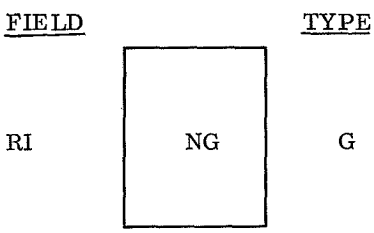
Most of the desirable features are provided by scheme 4, case a in that accumulator functions can be performed with base and index contents. Since it would be hard to over-emphasize the benefits in efficiency arising from being able to perform accumulator functions with base and index contents, scheme 4 is preferred when compared to all others except scheme 1; only the high cost of scheme 1 makes it hard to justify.

With scheme 4, as shown in Table 5 it can be seen that there are 15 registers. Accumulator functions can be performed with any register by giving its appropriate four-bit address in an R-field. Indexing functions are performed by those registers whose four-bit address is specified by explicit designation of the two low order bits in the X field of an instruction; the X field designation implies that the two high order bits of the four-bit address are "00." Base addressing is accomplished in a similar way with the two low order bits expressed in the B field, which implies that the two high order bits are "01."

When an index field (B or X) is used in an instruction, the two high order bits will be supplied by postprocessing the specified addresses in microcode prior to an actual reference. This is not too restrictive since it can likely be done before exiting the fetch sequence prior to interpretation. Although it does complicate the code, the additional flexibility afforded the macrolevel instruction repertoire and the associated reduction in main memory requirement is felt to outweigh the extra space, which should be negligible, required in ROM. In addition to flexibility, which has a negative effect on ROM size, we gain simplicity in format configuration requirements as shown below.

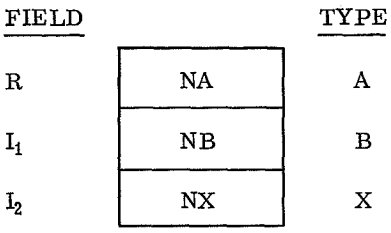
REGISTER FIELD FORMATS

To implement the scheme described above, four bits are required for an R-field and two bits each for a B and X field. From the point of view of R, B, X, and D fields, the configurations of Table 2 can be accommodated as illustrated in Figure 5 wherein the R, B, and X fields are determined completely. The associated Basic Instruction Set configurations that can be represented with these formats are shown on the right.



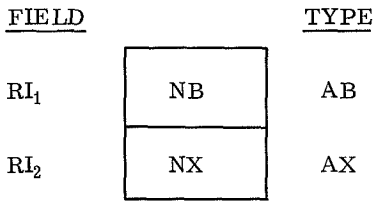
NB = NX = NA = NG

Scheme 1: All registers are type G and can be referenced in R- or I-fields. No registers are dedicated.



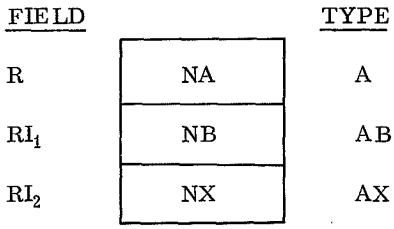
NA = NB = NX

Scheme 2: All registers are dedicated; A-type registers can only be referenced in an R-field, B in an I_1-field, and X in an I_2-field. There are no type G-registers.



NB = NX
NA = NX + NB

Scheme 3: Registers are grouped into two sets of semi-dedicated registers. AB-registers can be referenced in R- or I_1-fields and AX can be referenced in R- or I_2-fields. There are no general registers.



NX = NB
$NA = (NX + NB)^2$

Scheme 4: There are one set of dedicated A and two sets of semidedicated registers similar to those in scheme 3. There are no general registers.

Figure 3. Four schemes for register usage.

TABLE 3. BIT REQUIREMENTS AND NUMBER OF REGISTERS PROVIDED BY SEVERAL REGISTER ALLOCATION SCHEMES

Case	Number of Registers				Number of Field Bits			Bits	Total Registers
	G	A	B	X	R	B	X		
Scheme 1									
a	4	4	4	4	2	2	2	6	4
b	8	8	8	8	3	3	3	9	8
c	16	16	16	16	4	4	4	12	16
Scheme 2									
a	0	4	4	4	2	2	2	6	12
b	0	8	8	8	3	3	3	9	24
c	0	16	16	16	4	4	4	12	48
Scheme 3									
a	0	8	4	4	3	2	2	7	8
b	0	16	8	8	4	3	3	10	16
c	0	32	16	16	5	4	4	13	32
Scheme 4									
a	0	8	4	4	4	2	2	8	16
b	0	16	8	8	5	3	3	11	32

TABLE 4. NUMBER OF BITS FOR 16 REGISTERS

Scheme/Case	Number Bits	Number Registers
4/a	8	16
3/b	10	16
2/a-b	6-9	12-24
1/c	12	16

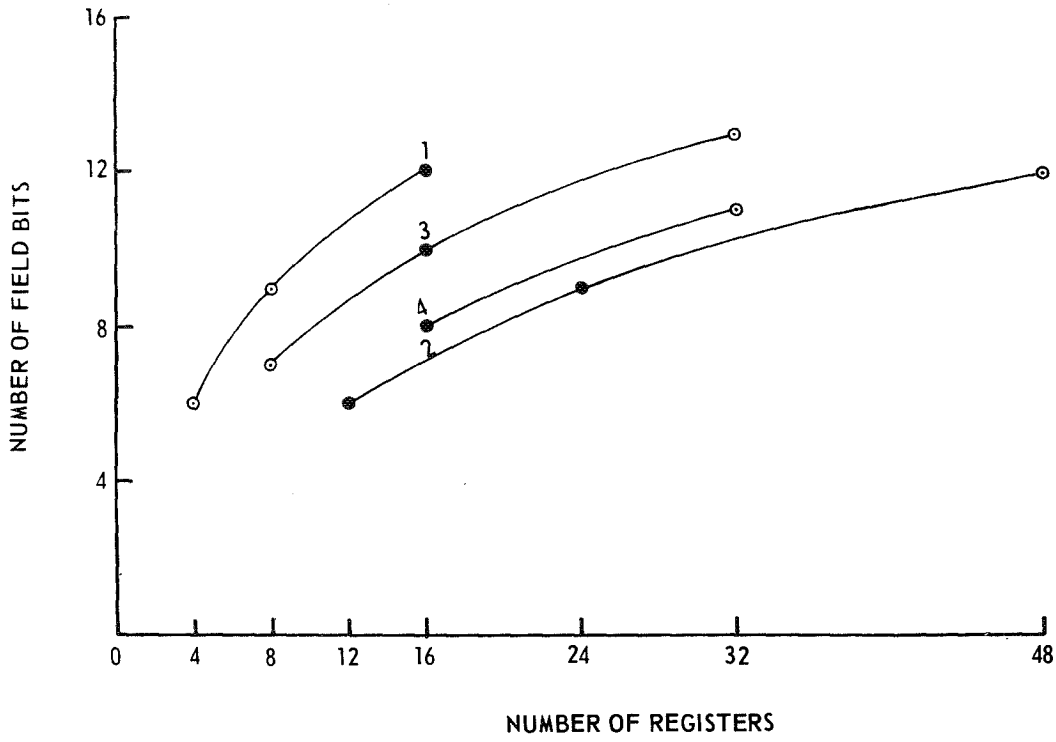


Figure 4. Bit cost per register for four schemes.

TABLE 5. REGISTER LAYOUT

Register Address	Function	Name
0 0 00	NO INDEXING	(0 or nil)
0 0 01	INDEX 1	X1 or R1
0 0 10	INDEX 2	X2 or R2
0 0 11	INDEX 3	X3 or R3
0 1 00	BASE 0	B0 or R4
0 1 01	BASE 1	B1 or R5
0 1 10	BASE 2	B2 or R6
0 1 11	BASE 3	B3 or R7
1 0 00	ACCUMULATOR 0	A0 or R8
1 0 01	ACCUMULATOR 1	A1 or R9
1 0 10	ACCUMULATOR 2	A2 or R10
1 0 11	ACCUMULATOR 3	A3 or R11
1 1 00	ACCUMULATOR 4	A4 or R12
1 1 01	ACCUMULATOR 5	A5 or R13
1 1 10	ACCUMULATOR 6	A6 or R14
1 1 11	ACCUMULATOR 7	A7 or R15

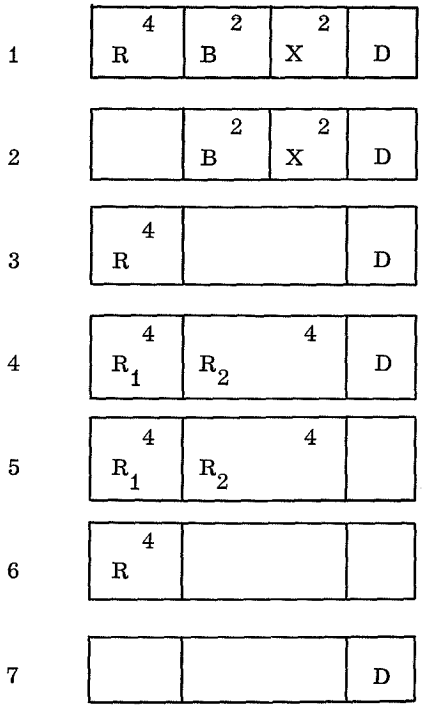


Figure 5. Partially complete formats.

Summary and Derived Repertoire

This section has shown how an alternate approach to register usage can effect a better blend of cost and flexibility. Factors other than register usage should certainly be considered but are beyond the scope of this discussion.

The final instruction set and associated formats are given below in summary form.

FORMAT 1

23 18	17 14	13 12	11 10	9 0
6	4	2	2	10
OP CODE	R	B	X	C

OP CODE	MNEMONIC	SHORT
01	L	Load
02	S	Store
03	LD	Load Double
04	SD	Store Double
05	LM	Masked Load
06	SM	Masked Store
07	SRR	Store Right Register
10	ORM	OR Register to Memory
11	OMR	OR Memory to Register
12	ARM	AND Register to Memory
13	AMR	AND Memory to Register

OP CODE	MNEMONIC	SHORT
14	ERM	XOR Register to Memory
15	EMR	XOR Memory to Register
16	ADD	Add to Register
17	SBR	Subtract from Register
20	MLT	Multiply to Register
21	DVD	Divide Register
22	SRLM	Skip on Register Less Than Memory
23	SREM	Skip on Register Equal to Memory
24	SRNM	Skip on Register Not Equal to Memory
25	SRGM	Skip on Register Greater than Memory
26	EXC	Execute
27	LEA	Load Effective Address
30	SAL	Set Alarm

FORMAT 2

23 18	17 14	13 10	9 0
6	4	4	10
OP CODE	R_1	R_2	C

OP CODE	MNEMONIC	SHORT
31	IRS	Increment Register and Skip
32	DRS	Decrement Register and Skip

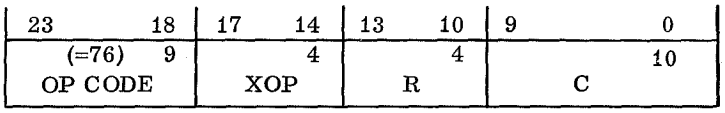
FORMAT 3

19 18	17 14	13 12	11 10	9 0
(=77) 6	4	2	2	10
OP CODE	XOP	B	X	C

OP CODE	XOP	MNEMONIC	SHORT
77	0	SP	Store P
	1	CLM	Clear Memory
	2	CM	Complement Memory

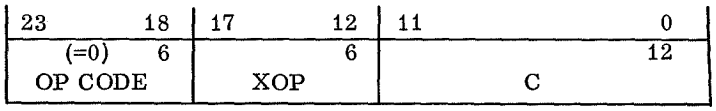
OP CODE	XOP	MNEMONIC	SHORT
	3	SMZ	Skip on Memory Zero
	4	SMN	Skip on Memory not Zero
	5	SMP	Skip on Memory Plus
	6	SMM	Skip on Memory Minus
	7	JP	Jump
	10	LJP	Load and Jump

FORMAT 4



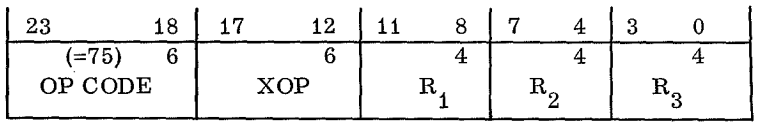
OP CODE	XOP	MNEMONIC	SHORT
76	0	RLC	Register Less than Constant
	1	REC	Register Equal to Constant
	2	RNC	Register Not Equal to Constant
	3	RGC	Register Greater than Constant
	4	IR	Increment Register
	5	LRI	Load Register Immediate
	6	R	Rotate
	7	RP	Rotate Pair
	10	S	Shift
	11	SP	Shift Pair
	12	LJA	Last Jump Address
	13	LRP	Load Register from P
	14	SIS	Sense Interrupt Status
	15	SIM	Set Interrupt Mask
	16	CIM	Clear Interrupt Mask
	17	CR	Complement Register

FORMAT 5



OP CODE	XOP	MNEMONIC	SHORT
00	00	HPR	Halt-Proceed
	01	NOP	No Operation
	02	RIP	Return from Interrupt Processor
	03	DSB	Disable Interrupts
	04	ENB	Enable Interrupts
	05	SSA	Set Stall Alarm
	06	RSA	Reset Stall Alarm

FORMAT 6



OP CODE	XOP	MNEMONIC	SHORT
75	0	SL	Skip Less
	1	SE	Skip Equal
	2	SN	Skip Not
	3	SG	Skip Greater
	4	TRR	Transfer Register to Register
	5	XR	Exchange Register
	6	AR	Add Registers
	7	SR	Subtract Registers

Page intentionally left blank

EFFICIENCY AND QUEUEING TIME CALCULATIONS FOR COMPUTER BANKS

By

B. G. Grunebaum

ABSTRACT

This paper describes mathematically the behavior of a set of computers connected in parallel to a queueing memory and fed by a probabilistic or rather stochastic sequence of programs. Possible underlying hypotheses are discussed and the ones that appear to be most practical from the standpoint of the system designer are used as the basis of closed analytical solutions for the calculation of the design parameters. Some specialized numerical procedures are indicated. Expectable variations of the basic problem and the mathematical modifications and/or additions that they imply are briefly discussed to show the applicability of these techniques in the design of more complex systems.

INTRODUCTION

Computing systems expected to fulfill requirements that are only known on a probabilistic basis are usually designed by simulation. This approach becomes rather cumbersome if the system is sufficiently complicated and/or if the number of variables involved is fairly large and varies over a wide range. A spaceborne computing system can be expected to fall into this category; and, in such cases, it is believed advantageous to use an analytical approach based on the knowledge of the behavior of individual subsystems. By isolating a sufficient number of distinct subsystems, they can be combined to any level of complexity.

The most frequently used subsystem, and probably the most difficult to analyze properly, is the simple system described in this paper. The results are presented without proof, because of space limitations; for additional details, see Reference 1.

FORMULATION OF THE PROBLEM

A bank of N identical computers is connected in parallel to a source of programs, which shall be refered to as the requestor (Fig. 1). The requestor emits an irregular sequence of execution requests, one request or call at a time. If, at the time of the call, at least one of the computers is not operating, one such computer will start executing; if all computers are operating, the request, i.e., the program called, will wait until the first termination occurs and then enter that computer. If another program is already waiting, the new request will queue sequentially. Every request will finally be handled by one and only one computer, unless the system is unstable by definition; i.e., the queue of requests keeps growing indefinitely. Some of the obvious refinements, not considered in this paper, are the existence of priorities, interrupt capabilities, and multiprocessing requests.

Answers are being sought for the following implied questions:

1. A stability criterion, in terms of the average running times of the programs and the probability of being called.

Furthermore, assuming the system to be stable:

2. The load factor L per computer; i.e., the average fraction of time each computer is operating.

3. The average processing time T_o; i.e., the average time it takes a computer to execute a program. The waiting time, if any, is not included in T_o.

4. The probability $q(t)$ that at least one computer becomes available in time t or less, if we interrogate at some random moment.

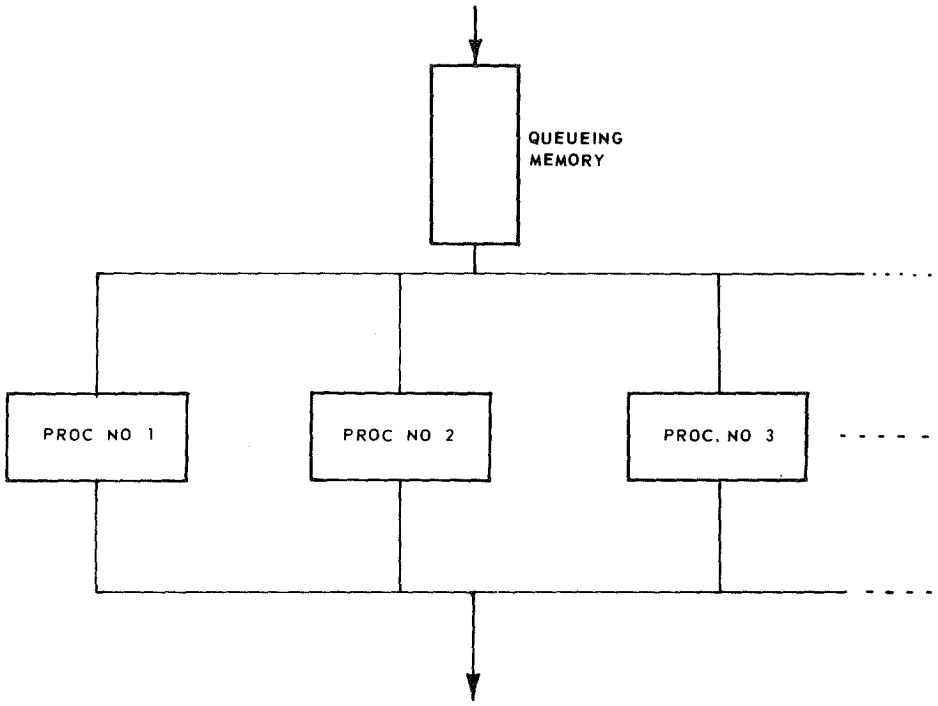


Figure 1. Basic subsystem.

5. The average waiting time T_1 for at least one computer to become receptive. In other words, if the computer bank is interrogated at some RANDOM moment, there is an average waiting time T_1 until at least one computer becomes receptive.

6. The probabilities $q_{i,k}(t)$ of having exactly k calls of the program u_i in the interval $(0,t]$, if the last preceding call of u_i occurred at $t = 0$.

7. The probabilities $Q_{i,k}(t)$ of having exactly k calls of the program u_i in a random interval $(0,t]$.

8. The probability $A_o(t)$ of having zero calls in a random interval $(0,t]$.

9. The probability $r_o(t)$ of having zero calls in $(0,t]$, if the last preceding (unspecified) call occurred at $t = 0$.

10. The probability a(t) that at least one computer becomes available in time t or less, if interrogation is at a moment when some computer starts executing a program and all other computers are known to be operating.

11. The probability a(t) that at least one computer becomes available in time t or less, if interrogation is at a moment when some computer starts executing a program and there is no information on the status of the other computers.

12. The probability $\phi(t)$ that a queue of non-zero length, i.e., containing at least one program, will remain unchanged during the next time interval of length t, if interrogation is at random.

13. The probability b(t) that a queue of constant non-zero length lasts at least time t.

14. The probability $\psi(t)$ that a queue of zero length lasts time t.

15. The average duration T_2 of a constant non-zero length queue.

16. The average duration T_3 of a zero length queue.

17. The average time T_4 that a program will remain in the same position in queue, if it arrived in a different position.

18. The probabilities P_k, $k = 0, 1, 2, \ldots$ that

we have a queue of length k at some random interrogation moment.

19. The average waiting time T_5 for any program; i.e., the average time between being called and being admitted to a computer for execution.

20. The maximum queue length h not to be exceeded with a specified probability.

RESULTS

The following definitions apply:

M = number of distinct programs u_i;

N = number of parallel processors;

t_i = average execution time of program u_i, $i = 1, (1), M$;

Q = time quantum (depends on experimental considerations and is otherwise arbitrary);

and

n_i = average number of times the program u_i is called in a time quantum $(0, Q]$.

The classical stability criterion is usually written in the form

$$T > NQ \quad , \tag{1}$$

where

$$T = \sum_{i = 1, (1), M} n_i t_i \tag{2}$$

and essentially says that the system is stable if, on the average, the working time requested is less than the computing time available. Nevertheless, it has been shown in Reference 1 that the condition (1) does not preclude the existence of finite probabilities for infinite waiting times and that in order to assure finite waiting times, the criterion (1) must be replaced by

$$nT_o \leq NQ \quad , \tag{3}$$

where

$$n = \sum_{i = 1, (1), M} n_i \quad , \tag{4}$$

and

$$T_o = \frac{1}{T} \sum_{i = 1, (1), M} n_i t_i^2 \quad . \tag{5}$$

Notice that formula (5) also represents the answer to question 3.

The load factor, as requested by question 2, is simply

$$L = \frac{T}{NQ} \quad . \tag{6}$$

To answer the next questions, it is necessary to assume that the programs have been indexed such that

$$t_o < t_1 < t_2 < \ldots < t_M \quad , \tag{7}$$

where $t_o = 0$.

Since the execution times t_i are measured values, it can be assumed that they are all distinct.

The answer to question 4 is now

$$q(t) = 1 - \left[1 - \frac{1}{T} \left(c_i + t \sum_{j = i, (1), M} n_j \right) \right]^N L^N \quad , \tag{8}$$

where the index i is determined from

$$t_{i-1} \leq t < t_i \quad , \quad i = 1, (1), M \tag{9}$$

and

$$c_i = \sum_{j = 0, (1), i-1} n_j t_j \quad , \quad i = 1, (1), M \quad . \tag{10}$$

Thus, $c_1 = 0$, $q(0) = 1 - L^N$, and $q(t \geq t_M) = 1$.

It is obvious that $q(0) < 1$, since one or more computers may be available at a random moment.

The answer to question 5 is

$$T_1 = \int_0^{t_M} t\, q'(t) dt \quad . \tag{11}$$

To present the answers to the remaining questions, the following definitions will be introduced:

$p_i(t)$ = probability that the interval between two consecutive calls of the program u_i is not greater than t,

$F_i(t)$ = probability that the interval between a random interrogation moment and the next call of the program u_i is not greater than t,

and

$$f_i(t) = F_i'(t) \quad .$$

The M functions $p_i(t)$ may be assumed known and the functions f_i and F_i can be determined with the aid of

$$f_i(t) = \int_0^\infty \frac{p_i'(t+\tau)\, f_i(\tau)}{1 - p_i(\tau)}\, d\tau \quad , \qquad (12)$$

a homogeneous Fredholm-type integral equation of the second kind, and

$$F_i(t) = \int_0^t f_i(\tau)\, d\tau \quad . \qquad (13)$$

It will be explained later how to determine the functions p_i from experimental data and how to solve equation (12) numerically. It must be mentioned at this point that, in general, the functions p_i are not even approximately exponential, so that the probability of a program u_i being called at some specific time does depend very much on the time of the preceding call of the same program and, perhaps, also on the times of other preceding calls. In other words, the oversimplification that the calling sequence is devoid of memory is avoided. Nevertheless, many of the forthcoming formulas assume the simplifying approximation that the probability of a specific program being called at some time depends at most on the time of the one preceding call of the same program. See Reference 1 for a detailed justification of this assumption.

The second assumption is that $M >> 1$ in the sense that the probability of any program being called at some specific time is very nearly independent of the time of the last preceding call. This restriction may, nevertheless, be dropped if all functions p_i are exponential; i. e., very unlikely to occur in practice.

Without any restrictions, the answers to questions 6, 7, 8, and 9 are

$$\left.\begin{aligned} q_{i,o}(t) &= 1 - p_i(\tau) \\ q_{i,k}(t) &= \int_0^t p_i(\tau)\, q_{i,k-1}(t-\tau)\, d\tau \end{aligned}\right\} \qquad (14)$$

$$\left.\begin{aligned} Q_{i,o}(t) &= 1 - F_i(t) \\ Q_{i,k}(t) &= \int_0^t f_i(\tau)\, q_{i,k-1}(t-\tau)\, d\tau \end{aligned}\right\} , \qquad (15)$$

with $k = 1, 2, 3, \ldots$ in formulas (14) and (15).

$$A_o(t) = \prod_{i = 1, (1), M} Q_{i,o}(t) \qquad (16)$$

and

$$r_o(t) = \frac{1}{n} A_o(t) \sum_{i = 1, (1), M} n_i \frac{q_{i,o}(t)}{Q_{i,o}(t)} \quad , \qquad (17)$$

with n specified by formula 4.

Notice that

$$\left.\begin{aligned} q_{i,o}(0) &= Q_{i,o}(0) = 1 \\ q_{i,k}(0) &= Q_{i,k}(0) = 0 \quad , \quad k > 0 \end{aligned}\right\} \qquad (18)$$

and

$$\left.\begin{aligned} q_{i,k}'(0) &= Q_{i,k}'(0) = 0 \quad , \quad k > 1 \\ q_{i,o}'(0) &= -p_i'(0) \qquad Q_{i,o}'(0) = -\frac{1}{T_{i1}} \\ q_{i,1}'(0) &= p_i'(0) \qquad Q_{i,1}'(0) = \frac{1}{T_{i1}} \end{aligned}\right\} , \qquad (19)$$

where

$$T_{i,1} = \frac{Q}{n_i} = \int_0^\infty t p_i'(t)dt \qquad (20)$$

is the average time between two consecutive calls of u_i.

Furthermore,

$$\left.\begin{aligned} A_o(0) &= r_o(0) = 1 \\ A_o'(0) &= r_o'(0) = -\frac{1}{T_6} \end{aligned}\right\} , \qquad (21)$$

where $T_6 = \frac{Q}{n}$ is the average time between consecutive calls, without specifying the nature of the programs. This constant is related to the function $r_o(t)$ by

$$T_6 = -\int_0^\infty t r_o'(t)dt \quad . \qquad (22)$$

The functions $r_o(t)$ and $A_o(t)$ may be generalized by the following definitions:

$r_k(t)$ = the probability that there will be exactly k calls in $(0,t]$ if there was an unspecified call at time 0,

and

$A_k(t)$ = the probability that there will be exactly k calls in a RANDOM interval $(0,t]$.

Then,

$$r_k(t) = -\int_0^t r_o'(\tau) r_{k-1}(t-\tau)d\tau, \quad k = 1,2,3,\ldots , \qquad (23)$$

$$A_k(t) = -\int_0^t A_o'(\tau) r_{k-1}(t-\tau)d\tau, \quad k = 1,2,3,\ldots . \qquad (24)$$

In particular,

$$\left.\begin{aligned} A_k(0) &= r_k(0) = 0 \ , \quad k > 0 \\ A_1'(0) &= r_1'(0) = \frac{1}{T_6} \\ A_k'(0) &= r_k'(0) = 0 \ , \quad k > 1 \end{aligned}\right\} . \qquad (25)$$

The function a(t) specified in question 10 is given by

$$a(t) = 1 - \left(1 - \frac{c_i}{T}\right)\left[1 - \frac{1}{T}\left(c_i + t \sum_{j=i,(1),M} n_j\right)\right]^{N-1} ,$$

$$M >> 1 \quad , \qquad (26)$$

with c_i specified by formula (10) and T by formula (2). The indexing must be such that the inequalities (7) and (9) are satisfied. Notice that $a(0) = 0$ and $a(t \geq t_M) = 1$.

The answer to question 11 is

$$a_1(t) = 1 - \left[1 - a(t)\right] L^{N-1} \quad , \qquad (27)$$

with a(t) specified by formula (26) and L by formula (6).

The answer to question 12 is

$$\phi(t) = A_o(t)\left[1 - \frac{1}{T}\left(c_i + t \sum_{j=i,(1),M} n_j\right)\right]^N ,$$

$$M >> 1 \quad , \qquad (28)$$

and the same conditions hold as for equation (26).

The function b(t) of question 13 is given by

$$b(t) = \frac{\frac{1}{T_6} b_1(t) + \frac{N}{T_0} b_2(t)}{\frac{1}{T_6} + \frac{N}{T_0}} \quad , \qquad (29)$$

where

$$b_1(t) = r_o(t)\,\frac{\phi(t)}{A_o(t)} \qquad (30)$$

and

$$b_2(t) = A_o(t)\Big[1 - a(t)\Big] \quad . \qquad (31)$$

The answer to question 14 is

$$\psi(t) = \psi_0(t) + \psi_1(t) + \psi_2(t) + \ldots \quad , \qquad (32)$$

where

$$\left.\begin{aligned} \psi_0(t) &= A_o'(t) \quad , \\ \psi_1(t) &= -\int_0^t \psi_0(X)\, a(x)\, \psi_0(t-x)\,dx \quad , \\ &\qquad M >> 1 \quad , \\ \text{and} \\ \psi_k(t) &= -\int_0^t r_o'(X)\, a_1(x)\, \psi_{k-1}(t-x)\,dx \quad , \\ &\qquad M >> 1 \quad . \end{aligned}\right\} \qquad (33)$$

The function $\psi(t)$ may also be determined with the aid of the integral equation

$$\psi'(t) + \int_0^t r_o'(x)a_1(x)\psi'(t-x)dx = A_o'(t)\Big[1 - a(t)\Big] + \int_0^t A_o'(t-x)\Big[r_o'(x)a_1(x) - A_o'(x)a(x)\Big]dx \quad , \qquad M >> 1 \qquad (34)$$

The answer to question 15 follows immediately from formula (29), namely,

$$T_2 = \int_0^{t_M} t\, b'(t)\, dt \quad . \qquad (35)$$

Similarly, the answers to questions 16 and 17 are

$$T_3 = -\int_0^{\infty} t\, \psi'(t)\, dt \qquad (36)$$

and

$$T_4 = \int_0^{t_M} t\, a'(t)\, dt \quad . \qquad (37)$$

The answers to the important question 18, namely the probabilities P_k, are given by

$$P_o = \sum_{m = 1,\ (1),\ \infty} \frac{T_3 g_{m,o} A^{m-1}(1-A)^m}{T_3 + (2m-1)T_2} \qquad (38)$$

and

$$P_h = \sum_{m = h,\ (1),\ \infty} \cdot \frac{T_2 g_{m,h} A^{m-1}(1-A)^m}{T_3 + (2m-1)T_2} \quad , \qquad h = 1, 2, 3, \ldots \quad , \qquad (39)$$

where

$$A = \frac{\frac{1}{T_6}}{\frac{1}{T_6} + \frac{N}{T_0}} \quad , \quad T_6 = \frac{Q}{n} \quad . \qquad (40)$$

The constant T_6 is determinable also with the aid of formula (21).

The constants $g_{m,h}$, $M = 1, 2, 3, \ldots$, $h = 0,\ (1),\ m,$ are mathematical constants whose meanings are further explained in Reference 2. They can be calculated with the following recurrences:

$$\left.\begin{aligned} g_{1,0} &= 1 \\ g_{n+1,0} &= \frac{2(2n-1)}{n+1}\, g_{n,o} \quad , \quad n = 1, 2, 3, \ldots \end{aligned}\right\} \qquad (41)$$

$$g_{n,1} = g_{n+1,0} \quad , \quad n = 1, 2, 3, \ldots \quad , \qquad (42)$$

and

$$g_{n,k} = \frac{k(n+1-k)}{(n+k)\ (k-1)}\, g_{n,k-1} \quad , \quad \begin{matrix} n = 2, 3, 4, \ldots \\ k = 2,\ (1),\ n \end{matrix} \quad . \qquad (43)$$

It should be noted that the stability of the system requires $A \leq \frac{1}{2}$, which can be shown to be equivalent to formula (3).

The average waiting time T_5, i.e., the answer to question 19, is now obtainable in the form

$$T_5 = T_1 P_o + T_4 \sum_{h=1,(1),\infty} \left(h + \frac{T_1}{L^N T_4} \right) P_h \quad , \quad (44)$$

with P_o and P_h given respectively by formulas (38) and (39). For control purposes, the inequality

$$\frac{1}{2} \leq \frac{T_1}{L^N T_4} \leq 1 \quad (45)$$

must be satisfied.

To answer question 20, simply observe that the probability for having a queue not longer than h programs is

$$Q_h = \sum_{i=0,(1),h} P_i \quad (46)$$

The maximum queue length h not to be exceeded with a specified probability c will be the greatest value of h such that $Q_h \leq c$.

NUMERICAL DETAILS

The suggested procedure for the determination of the functions $p_i(t)$ from experimental data is as follows.

For every program u_i, the times of arrival are measured over a sufficiently long interval of time. Let $t_1, t_2, t_3 \ldots t_n$ be the INTERVALS between consecutive arrivals of u_i, ordered so that

$$t_1 \leq t_2 \leq t_3 \leq \ldots \leq t_n \quad (47)$$

Then,

$$p(t_k) = \frac{k}{n} p(t_n) \quad (48)$$

where the index i has been dropped for convenience. If the measurements are sufficiently numerous, $p(t_n) = 1$ is assumed so that $p(t_k) = \frac{k}{n}$.

If some of the t_k's are equal, keep only the one with the HIGHEST index and drop the remaining ones from the list. Thus, the listing of the t_k's may contain less than n numbers but the highest index is always n. If s is the lowest index and k' is the index next higher to k, p(t) is obtained as a step function in the form

$$\left.\begin{aligned} p(t) &= 0 \quad , \quad 0 \leq t < t_s \\ p(t) &= \frac{k}{n} \quad , \quad t_k \leq t < t_{k'} \end{aligned}\right\} \quad , \quad (49)$$

where $t_{k'} = \infty$ if $k = n$. This is admittedly an approximation, but experience has shown that it appears to be much closer to the truth than any continuous function that could have been fitted to the information.

Assuming that the functions $p_i(t)$ are specified in the form (49), the Fredholm equation (12) is solved as follows.

Drop the index i for convenience as before and rewrite the inequality (47), after elimination of equal terms as explained preceding formula (49), in the form

$$t_{k_1} < t_{k_2} < \ldots < t_{km} \quad , \quad (50)$$

where

$$km = n \quad .$$

Then,

$$f(t) = \frac{1}{T_1} \left(1 - \frac{k_j}{n} \right) \quad , \quad t_{k_j} \leq t < t_{k_{j+1}} \quad (51)$$

$$j = 0, (1), m$$

$$k_o = 0$$

$$t_o = 0$$

$$t_{k_{m+1}} = \infty \quad ,$$

and

$$
\left.
\begin{aligned}
F(t) &= \frac{t}{T_1} \qquad , \qquad 0 \le t < t_{k_1} \\
&= \frac{1}{T_1}\left[\frac{1}{n}\sum_{j=1,(1),\ell}\left(k_j - k_{j-1}\right)t_{k_j} + \left(1 - \frac{k_\ell}{n}\right)t\right] \\
&\qquad\qquad t_{k_\ell} \le t < t_{k_{\ell+1}} \\
&\qquad\qquad \ell = 1,\ (1),\ m-1 \\
&\qquad\qquad k_o = 0 \\
&= 1 \qquad , \qquad t_{k_m} \le t \le \infty
\end{aligned}
\right\} \quad (52)
$$

As a final observation, let it be mentioned that special subroutines have been developed for the fast computational handling of the convolutions and other quadratures. It has not been found particularly difficult to modify this mathematical model for various requirements regarding priorities and interrupts.

REFERENCES

1. Grunebaum, B. G.: Efficiency and Queueing Time Calculations for Computer Banks. NASA Contractor Report, Contract No. NAS8-18405, July 1969.

2. Grunebaum, B. G.: The $g_{n,k}$ Numbers. Submitted for Journal Publication, Available from Computer Sciences Corporation.

BIBLIOGRAPHY

Grunebaum, B. G.: The Algebra of Probability Density Functions. NASA Contractor Report No. 61295, May 1969.

Grunebaum, B. G.: Random and Non-random Expectancies. NASA Technical Note, In Print.

APPLICATION OF DISCRETE DIGITAL SIMULATION LANGUAGES TO THE DESIGN OF ADAPTIVE DATA BUFFERING STRATEGIES

By

L. K. Paul, Jr.

ABSTRACT

Some of the design problems of computer-information networks are briefly discussed, with special reference to the anticipated requirements of a space-station/space-base system. The application of digital simulation techniques to the characterization of computer-information networks is described. Preliminary results of the discrete simulation of a data buffering scheme are presented.

DISCUSSION

Preceding papers have introduced the idea of a spaceborne computer system for an orbiting space station. Research efforts are currently underway at the Computation Laboratory of Marshall Space Flight Center that are designed to isolate and analyze the data management problems associated with such spaceborne computer systems.

The objective of this research is to identify data management problems by utilizing operations research techniques. One extremely useful operations research technique is the application of discrete simulation languages.

Therefore, the objective of this presentation is to introduce one of these simulation languages and illustrate how it is being used for designing spaceborne computer systems that will be capable of managing the data onboard a space station.

A space-station environment (Fig. 1) provides a unique setting for both the spaceborne computer and the data management system. This elaborate communications network consists of (1) the space station with its associated ancillary satellites, (2) the logistics shuttles, (3) the data relay satellite system, and (4) the ground-based command and control facilities. Each element of this network may need to communicate with every other element. Thus, the spaceborne computer system must perform its data management tasks within the context of a densely populated information network.

In addition to these external data management demands imposed upon a space station by its environment, there exist internal data management problems, many orders of magnitude more complex, which originate from within a space station itself.

For instance, the spaceborne computer system will be responsible for monitoring and controlling all operational systems onboard the space station. Budgeting of consumables, environmental control, malfunction detection, fault isolation, platform alignment, artificial gravity control, and verification of structural integrity will all require constant attention from the spaceborne computer. In addition, the spaceborne computer must collect, edit, and process data required for ephemeris determination, maneuver planning and execution, attitude stabilization, rendezvous targeting, and orbital traffic control. Even the scheduling of experiments as well as instrument setup, calibration, and checkout will be under the watchful eye of the spaceborne computer.

One way of gaining quantitative insight into these problem areas is by constructing a mathematical model of the system and subjecting this math model to extensive testing on the computer. Such experimentation is call "simulation."

To minimize the time required to code and checkout a math model, special simulation languages were developed. One of the most popular of these simulation languages is called GPSS, which stands for General Purpose System Simulation.

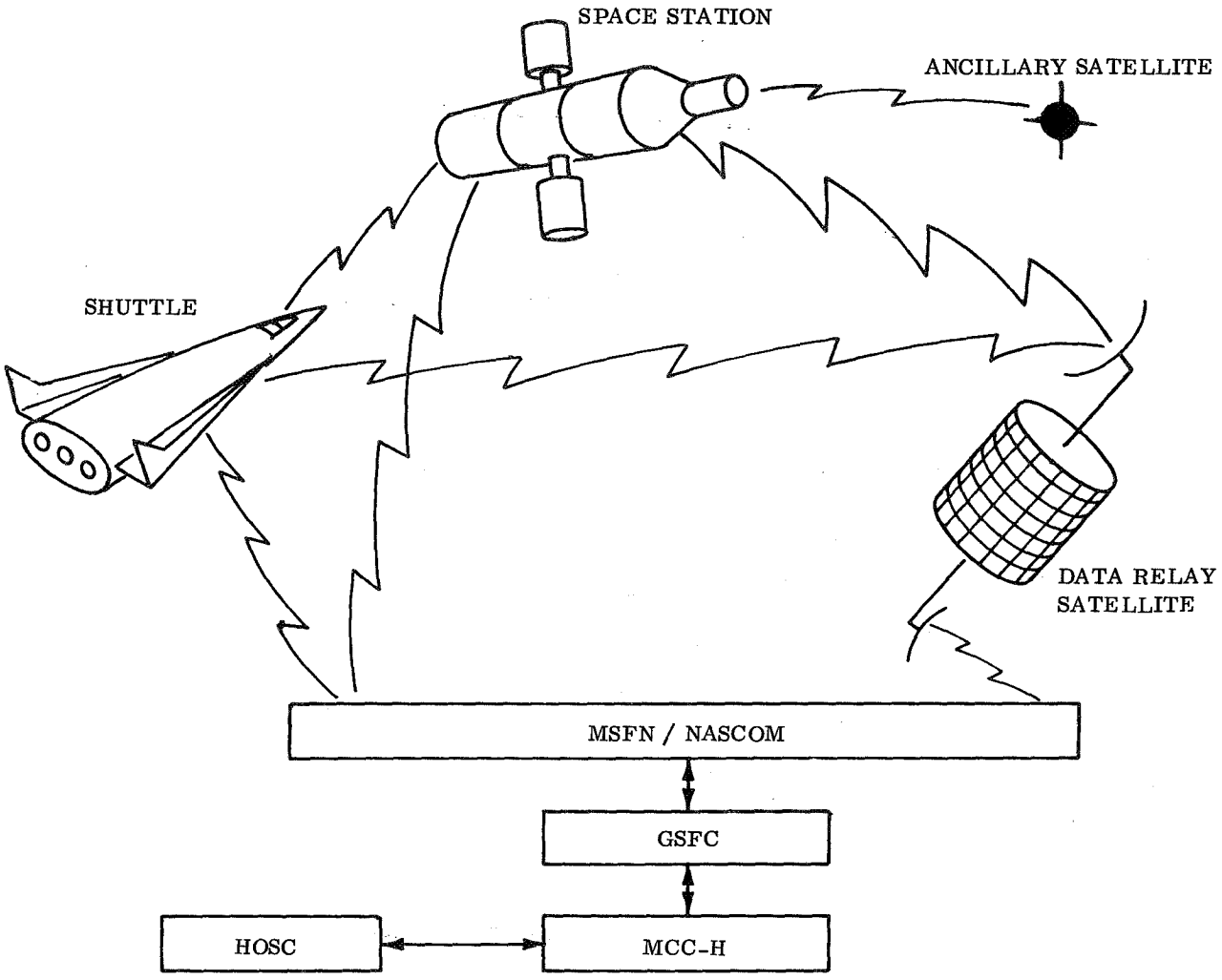
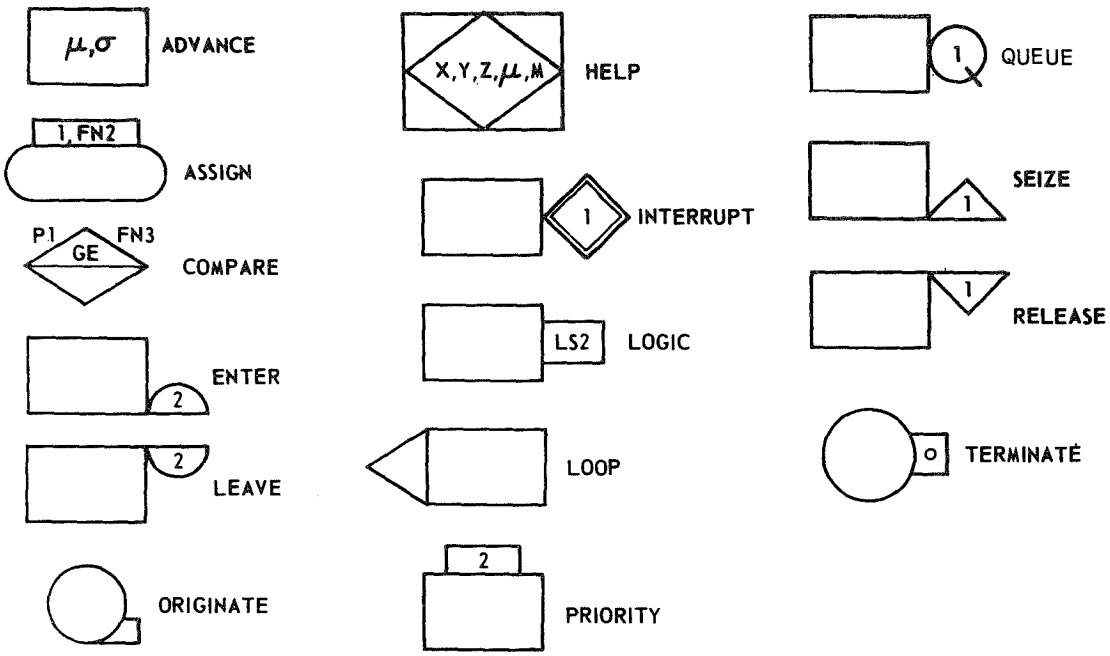


Figure 1. Space station environment.

The vocabulary of GPSS (Fig. 2) consists of over 50 prefabricated subroutines called "blocks" that can be selected and easily arranged by the systems engineer to represent his problem. The deck, consisting of one punch card for each block of the model, is submitted to the computer which, in turn, uses its compiler to complete the messy coding details and to produce a program that is completely checked out and ready to run. In general, modeling with GPSS is about four times faster than coding in FORTRAN.

Suppose that data arrive at a data compressor at a specified number of transactions per time unit (Fig. 3). The data compressor destroys the redundant data, and the worthwhile data continue to flow through the system until it arrives at a buffer. Here the data await their turn to be transmitted to the ground. The basic problem for such a system is that during periods of peak loading, the buffer may overflow. This may be prevented in several ways. The data transmission rate can be increased as the nonredundant data fill the buffer to 25 percent and 50 percent of its capacity. If the buffer becomes 75 percent full, a priority test can be imposed upon those data arriving for admission into the buffer. If the buffer continues to fill to 100 percent, all new data will be lost regardless of their importance. There are two basic logic loops; one that relates the data transmission rate to the buffer fullness and another that relates the buffer fullness to the priority required for admission. A communications engineer may wish to know such things as (1) how much data is redundant, (2) what percent of the data is lost because of a full buffer, (3) what percent is lost because of insufficient priority, (4) what is the



LOCATION	NAME (LEFT JUSTIFIED)	X	Y	Z	SELECTION MODE	NEXT BLOCK A	NEXT BLOCK B	MEAN TIME	MODIFIER	REMARKS
1 2	7 13	19	25	31	37	43	49	55	61	67 173
RANK	ASSIGN	1	FN2		BOTH	VIP	KILL			
VIP	COMPARE	P1	GE	FN3		QUE				
KILL	TERMINATE									

Figure 2. Abbreviated roster of GPSS II blocks.

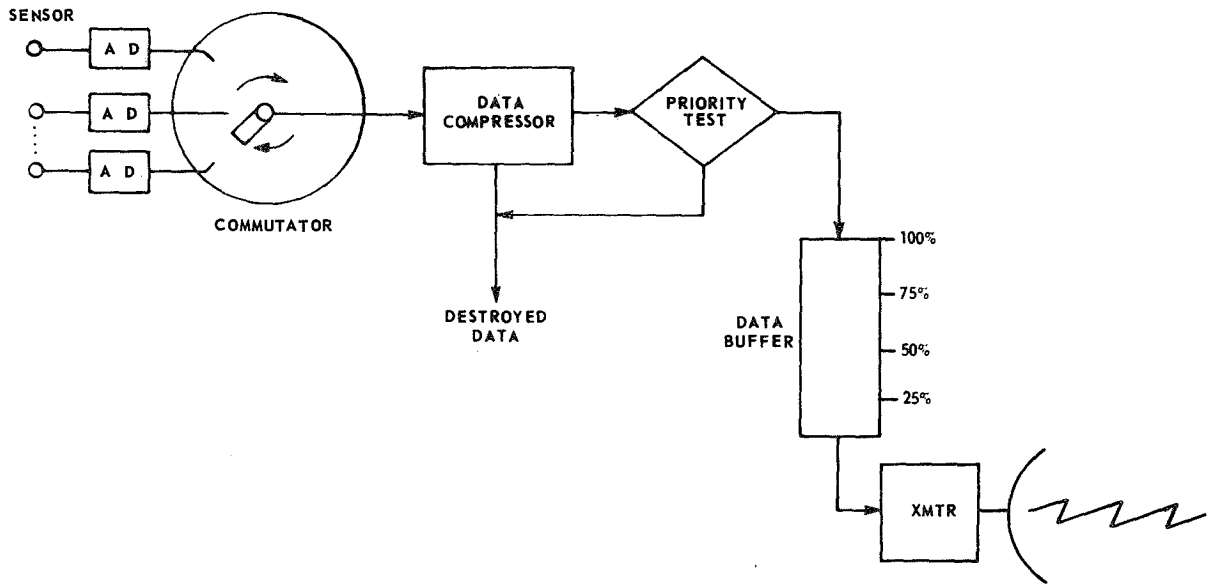


Figure 3. Typical information flow problem.

average buffer size, and (5) what is the average time spent waiting in the buffer?

This problem can be coded in GPSS as shown in Part A of Figure 4. By making full use of the power of GPSS, the problem can even be compacted into only three statements as shown in Part B of the same figure.

Part C of Figure 4 illustrates some typical results that may be produced by such a GPSS simulation, namely a time history of the buffer fullness. For instance, one would expect to see the buffer fill to 25 percent of its capacity at which point the data transmission rate would be increased. The buffer may continue to fill to the 50 percent point, and the transmission rate would increase to maximum. If the buffer continued to fill to the 75 percent level, a priority test would be invoked upon the incoming data. Under severe loading, the buffer could conceivably become saturated. Such information may reveal that after a period of 3 years a buffer of length L would be adequate, thus eliminating the requirement for priority electronics as well as cancelling the need for variable transmission capability.

CONCLUSION

This then is a typical example of an information flow problem and how it is coded in GPSS language, along with some traditional results that are available from such a simulation. From our research it is concluded that by using such operations research techniques as simulation, NASA will be able to design spaceborne computer systems that will be capable of managing the data onboard a space station.

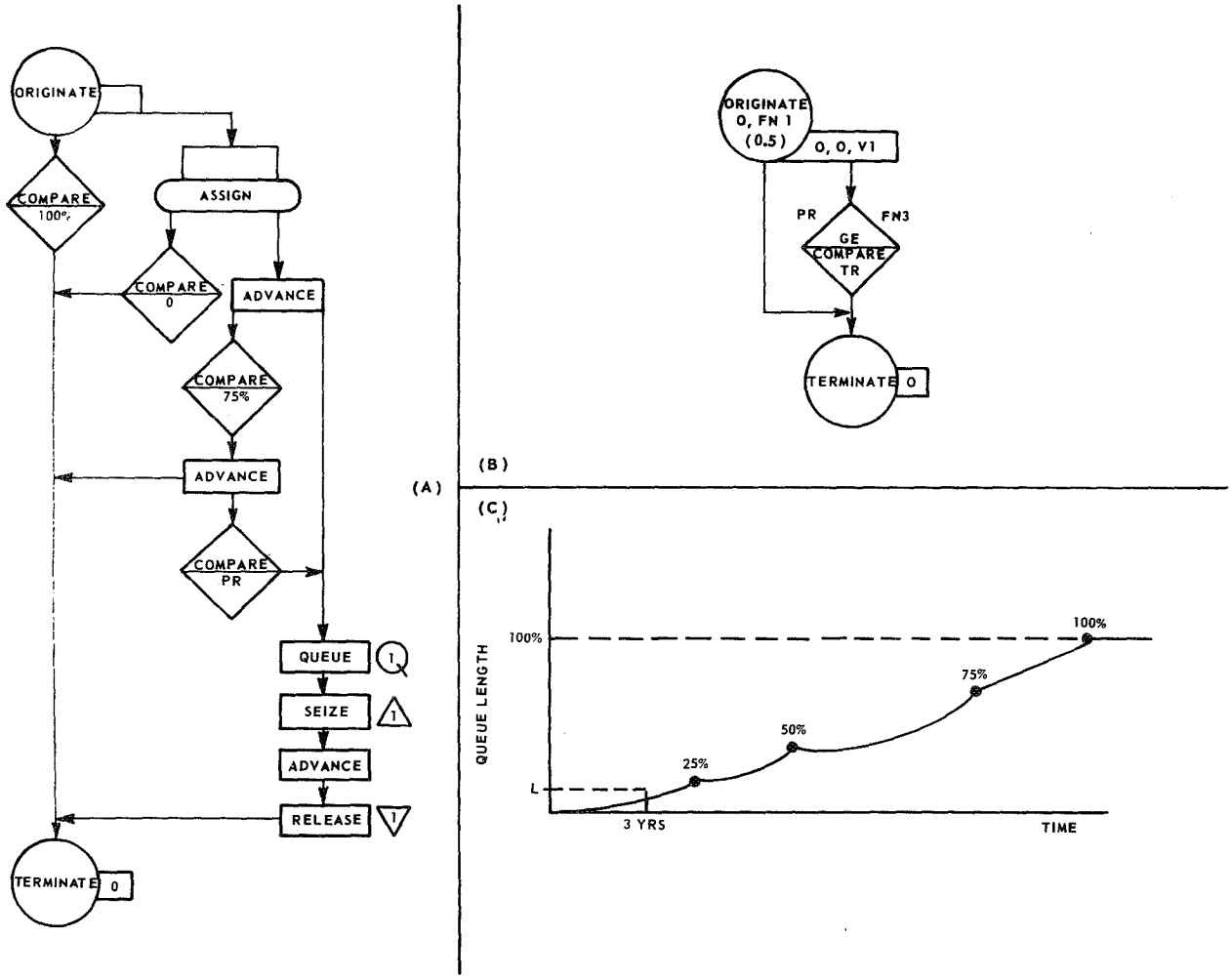


Figure 4. Buffer problem coded in GPSS.

BIBLIOGRAPHY

General Purpose Systems Simulator II. Reference Manual No. UP-4129, Univac Division of Sperry Rand Corporation.

Wallace, Gabriel R.: Buffer Fullness for a Data Compressor Using Saturn Vehicle Data. Research Achievements Review, Volume III, Report No. 5, NASA TM X-53810, 1968.

Page intentionally left blank

AUTOMATIC MALFUNCTION ANALYSIS (AMA) FOR DISCRETE SYSTEMS

By

D. T. Thomas and R. L. Jaegly

SUMMARY

The requirements for more automated checkout techniques and procedures have become more demanding as the complexity of aerospace systems increases. The original concept of checkout is changing from a semiautomated environment where computers are used to assist the test engineer in verifying performance of a system to a more autonomous environment where the computer is used not only to detect that a malfunction has occurred but also to isolate the cause of malfunction. Automatic Malfunction Analysis (AMA) is a tool that can be used in system checkout to verify testing procedures and determine failure candidates when a malfunction occurs.

The original effort on AMA was begun by General Dynamics Corporation, Convair Division, for fault isolation in the checkout of the Saturn IC stage at Marshall Space Flight Center (MSFC). The program, at that time, was written for the IBM 7094 and was successfully demonstrated in checkout of the engine cutoff system of the S-IC. The operational philosophy was to use an Automatic Test and Checkout Launch Language (ATOLL) Test Procedure written for execution on the checkout computer as the driving function. The basic functions of the procedures are to verify the systems performance and to control the checkout sequence. The ATOLL procedure, by comparing the predicted status of all discrete test points against the actual status, would detect a no-go condition when the predicted and actual profiles did not agree. Since the IBM 7094 was not on-line to the checkout computer, an analysis was made by AMA prior to the actual checkout of that system, assuming all possible combinations of no-go conditions at each point within the test procedure where the status of the discrete test points was to be checked. This precheckout analysis was made and the failure candidates for each possible no-go condition were generated on magnetic tape. Since the status of the system under checkout is maintained by AMA throughout the step by step simulation and analysis, the failure candidate tape generated on the 7094 can be accessed by the Test Procedure on the checkout computer, and failure candidates for any given discrete no-go condition can be determined. The results with this mode of operation were satisfactory; however, the time required to do the entire postcheckout simulation and analysis (5 to 7 hours) for one subsystem would be prohibitive in a normal checkout environment. To eliminate the excessive run time, the 7094 version was converted to the Univac 1108 and restructured so it could be accessed in near real-time from a remote checkout computer. In this case the checkout computer is an RCA-110A remoted via a Telpak A communication link.

Figure 1 illustrates the hardware configuration that is presently being used to further test AMA. It should be noted here that the use of the S-IC facility to develop the AMA techniques should not be interpreted as restricting AMA to this particular type of malfunction analysis. The availability of the facility, the RCA-110A, and the Univac 1108 simply afforded the ideal environment in which to develop AMA. In this hardware configuration, the system being checked out is still under control of the RCA-110A, where the Test Procedure is being executed. Once a no-go condition occurs, the number of the discrete test points that created the no-go condition are transmitted to the Univac 1108 for analysis by AMA. AMA, using a mathematical model of the system under test, will then evaluate the Boolean equations representing the system at the point where the no-go condition occurred. In the 1108 version, the Test Procedure is still used as the driving function for AMA.

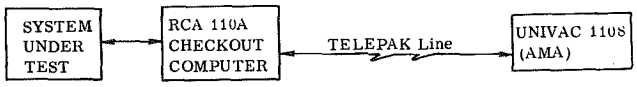


Figure 1. Hardware configuration.

By analyzing the data sent to the Univac 1108 by the checkout computer, AMA knows precisely

where the no-go condition occurred in the execution of the Test Procedure. AMA will then begin the analysis from that point. A state list is generated, and failure candidates for that particular malfunction are transmitted back to the checkout computer to be displayed on the Test Conductor's CRT display.

The on-line analysis allows the checkout computer to access AMA only when a malfunction occurs, thereby, drastically reducing the time required for analysis.

AMA FUNCTIONS

The analytical techniques in the Automatic Malfunction Analysis programs permit malfunction analysis to be made on a complex discrete system. These techniques can be broken down into six basic steps.

1. Prepare a model describing the network or system using logic or Boolean equations.

2. Analyze the equations to establish the interrelationships of the model.

3. Simulate real-time operation of the system using test procedures.

4. Using simulation, establish the system state at each point where analysis is to be made.

5. Analyze the system for each indicator to determine possible failure candidates.

6. Format data for display.

There are three primary programs required for Automatic Malfunction Analysis; however, Automatic Malfunction Analysis as used in this report refers to the total process which includes the following programs:

1. Preprocessor

2. Simulation

3. Automatic Malfunction Analysis

THE SYSTEM MODEL

The model is a series of Boolean algebra equations that logically describe the component interrelationships of a system network or partial system network. The Discrete Network Simulation (DNS) programs will chronologically simulate events occurring as a result of dynamic interactions among elements in a model. The use of the term, "Discrete Network," implies a system of variables defined in an interdependent relationship. Each variable is discrete as its action is a binary event in the network: on or off, acting or not acting, available or not available, true or false, 1 or 0, etc. The variables represent components, events, or activities. The network is described by: (1) a set of Boolean equations that completely define all interrelationships; and (2) a characteristic activity time associated with each variable (the time required for the effective change of state of a particular variable), which forms a complete mathematical model of a physical system.

Data for Model Building

To develop a model, the following information must be available:

1. A complete set of circuit diagrams for the system to be modeled.

2. Information that shows mechanical to electrical ties and vice versa, such as, limit and position switches for valves. If a written description is not available, all the related mechanical drawings will be needed.

3. Design specifications for relays, automatic swtiches, etc., where operating times are given.

4. An analysis prior to starting the model should be made to determine the boundary of the model, the number of variables that the model contains, and the inputs that will be necessary to make the model function.

5. The variables in a system to be modeled must be named prior to writing the model. Variables

can have any name as far as the computer programs are concerned (not to exceed 30 characters with current modification of the programs). The names should be easily identified on the documents from which they are written.

Boolean Equations and Time Parameters

The concept of using Boolean equations to describe electrical networks or other system intra-dependencies is well established. If individual time parameters can be attached to each element in a Boolean equation and evaluated, then a timed sequence description of a system operation can be generated. This section reviews the basic concept of using Boolean equations to describe an electrical network and describes the use of the parameter cards in the AMA programs.

Normally, all drawings show systems in a de-energized state. This applies especially to electrical drawings as indicated in the following circuit (Fig. 2). Where K1, K6, and K5 are relay coils, CK2, CK3, CK4, and CK5 are contacts operated by coils K2, K3, K4, and K5 which are not shown on the diagram. As shown, the 1-2 contacts of coils K2 and K4 are normally open when coils K2 and K4 are de-energized, and the 2-3 contacts of K3 and K5 are normally closed when coils K3 and K5 are de-energized. To energize coil K1, coil K2 must be energized, K3 must remain de-energized, and the bus must be energized.

Operators used in Boolean equations written to describe the hardware model are as shown in Table 1. An equation using the operators as indicated in Table 1 for K1 can be written as follows: K1 = K2*/K3* Bus. Equations are written from drawings that show the system in the de-energized state.

TABLE 1. BOOLEAN OPERATORS

OPERATOR	MEANING
=	Equal
.	Period
/	Not
*	And
+	Or

BINARY REPRESENTATION OF VARIABLES

Because every variable in the system can only be described as energized or not energized (VARIABLE or /VARIABLE, respectively) and since all variables are described in Boolean algebra, there are only two possible states for a variable. In Boolean algebra the two states are represented by a 1 and a 0. A 1 represents a signal, or energized, and 0 represents a lack of a signal, or not energized.

OPERATION OF A BASIC SIMULATION

Consider the simple circuit shown in Figure 3 as a system to be modeled. The equations that

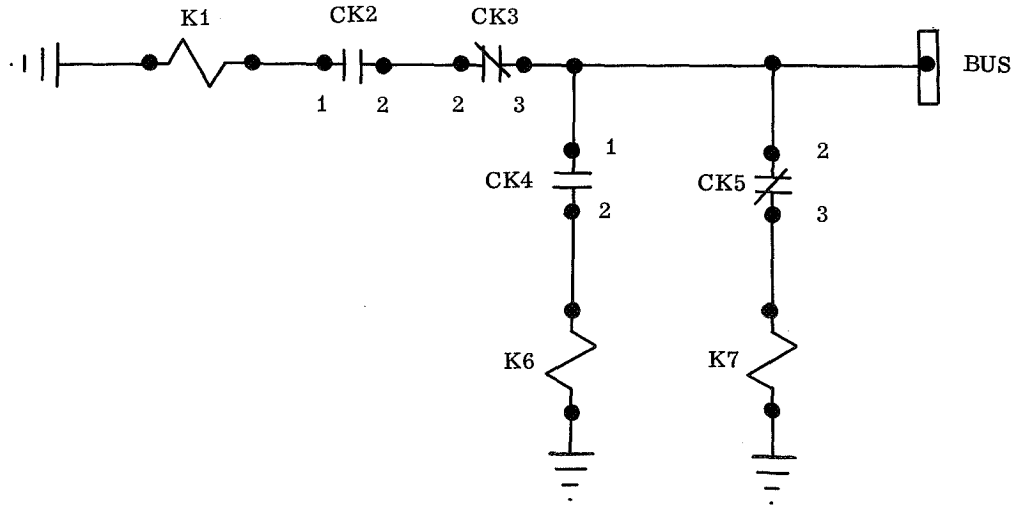


Figure 2. Typical relay schematic.

describe this circuit are:

K1 = Bus

CK1 = K1 ,

and

K2 = Bus * CK1 .

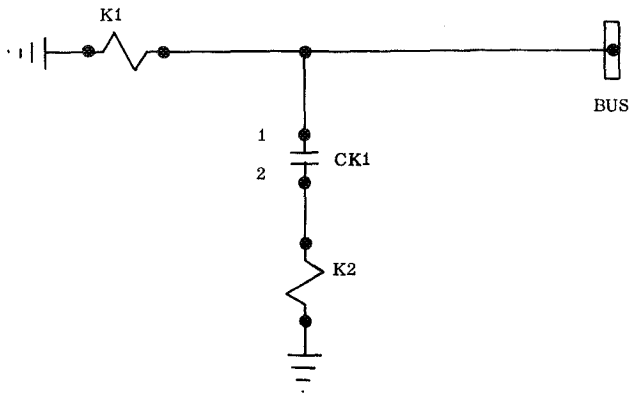


Figure 3. Simple logic equation sample.

In the static condition of the model, all variables are a binary 0. Changing this static model requires the change of state of some variable. To input the bus as a binary 1 would correspond to applying voltage to the bus in the hardware. When the bus becomes a 1, it will cause the coil K1 to become a 1 at a later time. This will depend on the operating time (pickup time) that was assigned to the coil. At a still later time, CK1 will become a 1, and, still later, K2 will become a 1. The time that these events take place depends on the operating time (pickup time) of the coil and the contacts. The model will then stay in this condition; that is, all variables are a binary 1. If the bus is input to a binary 0, all the variables in the model will go to 0, just as would happen in the hardware if the bus is de-energized. The time of deactivation will depend on the dropout time assigned these variables. These are normally the times shown on the design specifications for the different variables. The pickup and dropout times required for each variable are entered in the model via the parameter card.

PARAMETER CARD

A parameter card is required for each variable in the model and is used for three different functions.

1. The parameter card is used in the equation compiler program to simplify the writing of the logic equations.

2. The parameter card is used in the AMA programs as the source of the activation times and as classification codes for each component in the model.

3. The parameter card can be used to store information related to each variable in the model that can be used for future reference. This information is not used in any way by the AMA programs.

The parameter card contains the variable name, from 1 to 6 time fields, each containing 6 characters, and up to 40 data fields, each containing not more than 42 characters. For use with the AMA programs, all the parameter cards associated with a single model must contain the same number of fields. The fields are delineated and labeled by a control card used with the AMA preprocessor program.

On the individual parameter cards, following the variable name, the fields are designated by data followed by a comma or a blank followed by a comma. The end of the data field string is terminated by the use of a period. This is defined further in the preprocessor program description.

The time fields, which are used in system simulation, define the activation times for each component in the model. Up to six time fields may be defined, and any two from these six may be used in the system simulation to represent activation and deactivation times respectively. The time base for each variable may be individually specified and can cover the full range from microseconds to days. On the parameter card, the time units can be specified using up to five digits and a base designated by a single letter.

Information placed in the data fields at the time of model preparation classifies the variables as active or inactive in terms of the system simulation and catalogs the variables into classifications relating to the electrical or physical description of the variable. This classification is used in the AMA

programs to determine which components should be considered as possible malfunction candidates and under which state (on or off) they could be malfunction candidates.

Writing Equations

This section describes the techniques that can be used to prepare or describe the system that is to be simulated and/or analyzed. The AMA program will process any set of Boolean equations. The accuracy of the simulation and/or analysis is a function of how accurately the set of Boolean equations can describe a given system. The examples given and the principal applications of the AMA programs have all been based on electrical networks, but this is not to imply any limitation in the programs themselves to handle only electrical networks. Any set of Boolean equations could be processed.

The following example including the rules and guidelines stated therein have been found to be effective and accurate when used in conjunction with the AMA programs to permit a system simulation and subsequent malfunction analysis. They can be used and expanded to describe other types of systems for which simulation and analysis is desired.

SIMPLE SERIES CIRCUIT

Equation (1) is written as follows to describe the simple series circuit shown in Figure 4.

$$K1 = CK3*/J7*/CK4*/J6*Bus \quad (1)$$

Equations are normally written from the power consuming elements to the power source. In this example K1 represents a coil, CK3 the normally open contacts of another coil whose circuit is not shown, J7 a pin jack connection through which current can normally flow, CK4 the normally closed contacts of another coil whose circuit is not shown, J6 another pin jack connection, and finally the bus represents the power source.

The negation sign preceding the variables J7, CK4, and J6 indicates that there is normally a completed circuit through these variables when the pin jacks are connected and through CK4 when CK4 is de-energized.

The negation sign, or 'not' operator, is normally used to represent a de-energized state. The use of the negation sign in front of variables representing pin jacks in other inactive type components represents a compromise between a true system description and the mechanics of the AMA programs. The pin jacks and other passive type components such as fuses normally stand by themselves in the logic equations. They do not appear in any cause and effect relationship. To describe a completed circuit, each of these passive components would have to be set to a 1 value by an external input to the simulation portion of the AMA program. The use of the negation sign causes the 0 or de-energized state of the component to appear as a 1 in the total program, thus eliminating the need for many arbitrary inputs to the simulation program for the passive type of components.

PARALLEL CIRCUIT

Equation (2) describes the parallel circuit shown in Figure 5.

$$K1 = /CK4 * CK2 * Bus + /CK3 * CK5 * Bus \quad . \quad (2)$$

Equations that describe the parallel circuits can be written using parentheses as shown in equation (3). However, if the model is to be used for Automatic Malfunction Analysis, equations containing parentheses cannot be processed by the current version of the AMA program.

$$K1 = (/CK4 * CK2 + /CK3 * CK5) * Bus \quad . \quad (3)$$

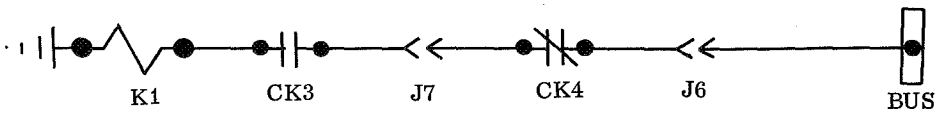


Figure 4. Typical series circuit.

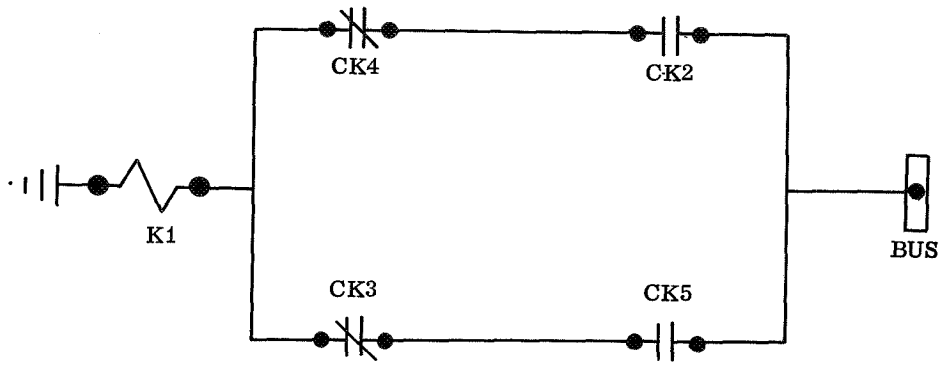


Figure 5. Typical parallel circuit.

PREPROCESSOR PROGRAM

The Discrete Network Simulator Preprocessor is a versatile data reduction program. It is used for reformatting, coding, and filing simulation model data into various formats as desired by the user. The program was designed to shorten and reorder model equations and names so that subsequent programs are not required to utilize the model data in its original format of name/parameter and equation card images. The names in the original model are descriptive engineering names that are necessary for accurate correlation between the model and the system. The model data must be coded and reduced for fast and efficient internal computer manipulation. The preprocessor program accomplishes this function and records the processed model data on files that provide rapid direct usage by the simulation and AMA programs. The capability for subsequent programs to print out the processed data with engineering names still is possible through special print processing subroutines. The output files are stored on three separate tapes, plus a standard model listing with error identification. One output tape file is formatted for use by the simulation program and one is generated in a format for use by the AMA program. The first tape is the standard preprocessor output tape and can be produced without generating either of the other output tapes. This tape must be used as an input tape to the preprocessor program for constructing either one or both, as desired, of the other output tapes. The Standard Model Tape also is utilized as an input tape to the preprocessor editor program to further process the model and produce cross references and interdependency tables for analysis.

The data are extracted for processing as outlined by the field assignment or * VARIABLES control card. The total number of fields used in conjunction with each variable is tabulated for use during processing. The active variables (variables that can change state during system operation) and those that are inactive are separated and counted. these totals are listed on the printout for reference. During the initial processing, the variables are assigned internal numeric codes that are represented by their implied position in the active or inactive names tables. The active names will start at table location 1 to N and the inactive names will be from N + 1 to the table end.

As the equations are processed for use by subsequent programs, they are packed into equation tables containing coded (shorthand) versions with extraneous blanks, etc., removed. The variables used in the equations are negated before being packed into the tables. For the simulation model, the equations are searched, and the active variables that affect other variables by their use in the equations are tabulated and their relationship to each other is identified. Reference tables reflecting this relationship are generated. An Indirect Reference table lists the total number of variables that each individual variable directly affects. This table also provides an address that points to a location in a Direct Reference table that contains the locations of the equations for the affected variables in a separate equation table. In addition to the reference and equation tables, a model is created in which the inactive variables are removed for the simulation program.

This model has the equations reformatted, or culled of the inactive variables, and is constructed of the internal code numbers. The actuation times are cataloged into files that can be readily identified with their associated variable during simulation execution. For Automatic Malfunction Analysis, each name is assigned a computer word by implied position as before, only this new table will contain all of the characteristics of the variable indicated by Bit flags. The characteristic will include the classifications for: active or inactive; whether the variable is to be considered as a failure candidate including which type of failure, on or off; the variable common point level, or the number of indicators the variable can affect; and the starting address of the variables equation. A failure names table is also created that lists all of the variables designated as failure candidates and with a corresponding search key section that will eliminate unnecessary testing of variables during the AMA program execution.

By the use of control cards, several options in processing the model are controlled. Function of the *VARIABLES control card is to define the variable parameter card formats. The card has a free field format and must occur before the variable parameter cards. Two required pieces of information must be obtained from the card: (1) the number of fields associated with the variables, and (2) which fields are time fields. The fields may be assigned a label or left blank and referenced by the relative position on the card.

The number of fields associated with the variables is determined by the number of commas plus 1. The field signifying the time fields must have the word TIME between the commas (or comma and terminating period). Field 1 is always considered to be the variable name.

The *CULL card controls the equation processing. All variables that are not classified as active as defined by the CULL card are deleted from the equations.

*CULL, FIELD TITLE = IDENTIFIER.

(A) *CULL is placed behind *VARIABLES card and before any time cards.

(B) FIELD TITLE = IDENTIFIER.

FIELD TITLE is the actual name assigned to identify the field position of the information controlling the active-inactive designation.

IDENTIFIER is the BCD code that identifies an active variable card.

Example:

*CULL, TYPE = A.

Field title is TYPE, and identifying code is A. Thus, *VARIABLES, TYPE control card indicates that the first data field contains active-inactive code, and any variable card with an A in the first field is treated as active. Automatic Malfunction Analysis control cards define the information in the AMA model for each variable.

*FAILURE CLASS, title. Identifies the name of the variable definitions field that will classify failure candidates.

*ZEROS, codes. Indicates the zero failure classes. Codes are alpha-numeric values in field designated by the failure class card that will identify a zero candidate. Each code is separated from the previous by a comma, and the card terminates with a period. Maximum of 50 codes.

*ONES, codes. Same as *ZEROS, except the data concerns one's failure candidates.

*INDICATORS, title = codes. Card identifies the monitored variables in an AMA system. Title is the name of the variable definitions field that contains the identifying code for indicators. Codes are alpha-numeric values in field designated by the title that will identify an indicator. Maximum of 50 codes.

*LOCATION, field 1, field 2. Card identifies two variable definitions fields which contain additional data to be included when failure candidates are listed.

Discrete Network Simulation Program

After the system model has been processed through the preprocessor editor programs, system operation can be simulated using the Discrete Network Simulation (DNS) program. The model, as written, represents the system in a static condition. A set of drive functions (test procedures) is then required to establish the model in the initial condition for the simulation and to represent the activities to be simulated. The results of the simulation are recorded on the output tape. The printout includes:

1. The order of events occurring.

2. The time of each occurrence.

3. A list showing the state of all variables at any selected time.

The program first establishes an initial condition for the state of the variables in the model. On the basis of this state, it examines all equations in the model, and the logic predicts what variables will change state. The simulation represents real time; therefore, after the prediction has been made, the program looks up the activation times for the variables changing state and imposes that time delay on the program before allowing the predicted changes to occur. This process is indicated on the printout. "Input" is the code word indicating that the state of this variable is being set by an external command in the input deck. The code word, "Enter" indicates that this variable is changing state because of the logic of the equations combined with the computer program.

This simulation represents real-time. The time of the simulation is printed in the columns on the right side of the sheet and as indicated by the headings, can be described in days, hours, minutes, or seconds. The seconds are resolved down to the nearest millisecond. The time printed for each activity line represents the actual time for the activity to occur.

Figure 6 is a typical printout from the simulation program. The printout shows the complete operational history of the system being simulated. The procedure identification is IDA 5010. At step number 5, substep 00, when DO 179 is input to 1, it predicts that relay 190K5 will come on. Relay 190K5 comes on and predicts that DI's 176, 177, and 178 and relays 27K43, 44, and 45 will turn off. Following the predictions, the actual events take place and the absence of any predicted results shows that no further activities take place as the result of DO 179 being turned on.

At the end of the simulation of each section of the procedure, the simulation program is commanded to print out a list showing the status (on or off) of each of the components (variables) in the system; an example is shown in Figure 7. This indicates, at the end of the procedure, which relays were on and which were off at this point in the procedure. In the simulation program, a record has been kept

```
*
*HEAD LIST
*       *NAME          IDA5010

TYPE            DESCRIPTIØN              VALUE  NØ. ØF EVENTS   SECØNDS

*BEGIN,10000.
*       *STEP NØ    000500
INPUT           DØ179                    1      1               .100
                190K5                    1                      .105
ENTER           190K5                    1      1               .105
                DI176                    0                      .105
                DI177                    0                      .105
                DI178                    0                      .105
                27K43                    0                      .110
                27K44                    0                      .110
                27K45                    0                      .110
ENTER           DI176                    0      6               .105
ENTER           DI177                    0      5               .105
ENTER           DI178                    0      4               .105
ENTER           27K43                    0      3               .110
ENTER           27K44                    0      2               .110
ENTER           27K45                    0      1               .110

THERE WERE          8 EVENTS IN CASE     45.
```

Figure 6. Simulation history printout.

EDS TEST PRØCEDURE VERIFICATIØN

149 602A5K69	=	0.	130
150 602A5K70	=	0.	134
151 602A5K71	=	0.	132
152 602A5K78R	=	1.	0
153 602A5K78S	=	0.	0
154 602A5K40	=	1.	0
155 602A5K41	=	1.	0
156 602A5K42	=	1.	0
157 602A5K65R	=	1.	0
158 602A5K65S	=	0.	0
159 602A5K66R	=	1.	0
160 602A5K66S	=	0.	0
161 602A5K11	=	0.	6
162 602A5K12	=	0.	6
163 602A5K13	=	1.	7
164 602A5K14	=	0.	8
165 602A5K21	=	0.	10
166 602A5K22	=	0.	10
167 602A5K23	=	1.	11
168 602A5K24	=	0.	12
169 602A5K31	=	0.	4
170 602A5K32	=	0.	4
171 602A5K33	=	0.	4
172 602A5K34	=	0.	4
173 602A5K77	=	1.	69
174 602A5K8	=	1.	91
175 602A5K15	=	0.	8
176 602A5K16	=	1.	9
177 602A5K17	=	0.	10
178 602A5K18	=	0.	10
179 602A5K25	=	0.	12
180 602A5K26	=	1.	13
181 602A5K27	=	0.	14
182 602A5K28	=	0.	14
183 602A5K35	=	0.	4
184 602A5K36	=	0.	4
185 602A5K37	=	0.	4
186 602A5K38	=	0.	4

Figure 7. State list from discrete network simulation.

of the history of the activities of each of the variables in the system, as shown in the right hand column. A zero in this column indicates that these relays have not been turned on or off in this portion of the procedure. It also shows that relay 602A5K69 was turned on or off 130 times during this procedure. Each count represents a half cycle which results from that element being turned on or off. If the figure in this counter happens to be an odd number (such as the occasion for item 163 relay A5K13), it indicates that this relay is left in the opposite state at the end of the procedure compared to what it was at the beginning of the procedure.

From these examples it can be seen that considerable information about the effect of a given test procedure upon the hardware or system under analysis can be deduced by careful evaluation of results of the system simulation — provided a carefully prepared model is used.

Test procedure verification using the Discrete Network Simulation program is diagrammatically shown in Figure 8. Using the schematic, a logic model of the electrical networks is constructed. From the test procedure tape, the commands to the stage and the electrical support equipment are

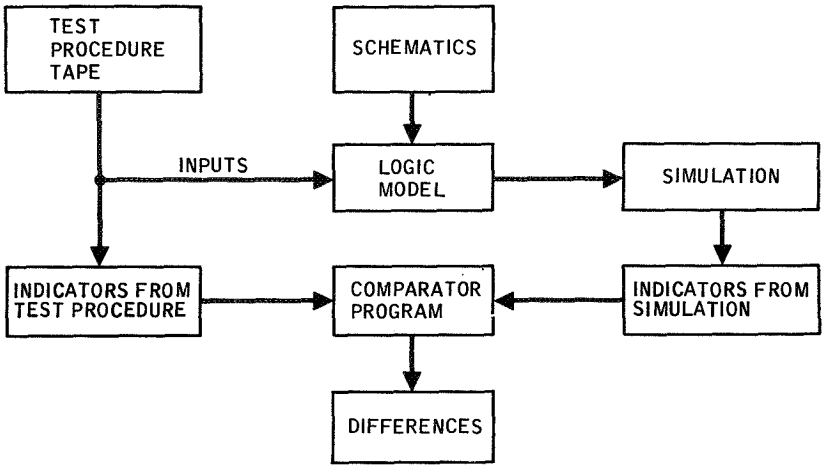


Figure 8. Test procedure verification by discrete network simulation.

abstracted and these, together with the model, make up the input to the Discrete Network Simulation program.

The simulation program produces a time oriented history of the normal sequence of operation of the electrical networks. The complete history of the normal network operation is used to validate the accuracy of the logic model. However, the checkout computer can only evaluate the results of inputs it has supplied in terms of discrete indicators. Therefore, an edited version of the simulation history is produced that shows only the commands from the computer and the results that are transmitted back to the computer.

The test procedure tape, in addition to the command functions, normally contains the predicted results for the procedure. These predicted results are abstracted from the test procedure tape and compared to the results of the discrete network simulation by a comparator program. Differences between predicted results and the test procedure and results of the simulation are printed out for each step and substep of the test procedure.

AUTOMATIC MALFUNCTION ANALYSIS

Automatic Malfunction Analysis is a comprehensive analysis technique to determine component failure that will affect designated test or monitoring points that indicate the status of equipment. This technique is based on an orderly analysis process to determine the failure candidate(s) that would cause specific no-go test-point patterns for each test point in a test procedure.

Functional failures can occur individually, in the sense that a failure is independent of other failures. An independent failure of a component is considered a single failure — an occurrence AMA is designated to detect. A single failure can effect no indicators, effect a single specific indicator, or effect multiple indicators. Because AMA is comprehensive, data are generated to encompass all possible single failures that would have any single or multiple effects.

In use, the on-line AMA must have as inputs from the test terminal suitable formatted input data that include:

1. The location within the test where malfunction or no-go occurred. This is normally the block, step and substep, etc. of a particular procedure.

2. The indicators or DI's that are in an incorrect state or no-go.

Upon receipt of this data, the test location data are stored and the simulation starts. Each of the variables are initialized to their test beginning states and an internal Block Initialization Test is accessed to update the state lists to the correct test block. The simulation will then continue through the test block of the procedure until it reaches the point (step and substep) that corresponds to the one sent from the test console. After the simulation is

complete, the states of the variables at this point are added to the internal tables. The automatic malfunction subprograms will analyze the relationship of each variable within each equation as determined by the normal state of each variable at this block, step, and substep. The results of this analysis produce a cause and effect classification of all variables in the system for the time that the no-go occurred. The causes are compared with the no-go indicators that were input from the test console, and if an exact match or pattern of DI's is found that could be effected by a particular variable (from the cause and effects table), these variables will be uptranslated to do their original engineering format and returned to the test console for display. The display will include such information as necessary to identify the variable clearly and indicate where it may be found on the schematics. If an exact match is not found, the nearest group of single failures that falls within the no-go set is flagged as the cause with notation to this effect.

Figure 9 is an example of the information displayed on the CRT display of the checkout computer as the result of requesting AMA data. This information was generated on the Univac 1108 and transmitted to the checkout computer. The information that was transmitted to the 1108 is sent back and displayed to check that the correct analysis was performed.

1. The Test Procedure is 321.

2. AMA data is requested at Block 2, Step 1, Substep 82 of this procedure. The following information was returned:

1. DI 1000 was not included in the logic model, therefore no information is available.

2. The state of DI 921 as transmitted agrees with the normal state as defined by the system simulation. Therefore, it is not a valid no-go.

3. There are three DI's, 855, 856, and 859 processed as no-go's.

4. There were three malfunction candidates; any of the three if they failed would have caused the multiple DI no-go pattern.

5. If each DI were considered a separate independent failure effect, then there is one component for each DI which could cause that no-go.

```
DNS/AMA DATA -   TP321,    BLOCK 2,        STEP 1,          SUBSTEP 82.

THE FOLLOWING, NOT IN THE MODEL, WILL BE IGNORED . . . . .
      DI1000

SIMULATION INDICATES THE FOLLOWING IS NOT A VALID NO-GO . . .
      DI921

THE FOLLOWING NO-GO DI LIST WAS PROCESSED . . . .
      DI855          DI856          DI859

MALFUNCTIONS EFFECTING THE MULTIPLE NO-GO DISCRETES ARE -
1587 CONTACT'12A3'K4'J8SHSI              GSE              294
 769 BUS'12A3'21D110                     GSE              294
1137 COIL'12A3'K4'J8SBSC                 GSE              294

MALFUNCTIONS WHICH EFFECT THE SINGLE DISCRETE DI855 -
1577 CONTACT'12A3'K1'J8GH                GSE              294

MALFUNCTIONS WHICH EFFECT THE SINGLE DISCRETE DI856 -
1580 CONTACT'12A3'K2'J8PR                GSE              294

MALFUNCTIONS WHICH EFFECT THE SINGLE DISCRETE DI859 -
1582 CONTACT'12A3'K3'J8SAZ               GSE              294

                                                          ENDATA
```

Figure 9. AMA display.

6. For every component listed the following information is displayed.

a. The complete name as written in the equation.

b. The internal code number which could be used to call up additional information about the component if it were included in the model data base.

c. The location of the component as to stage, GSE, or facility.

d. The sheet number of the schematic where that component is used.

COMPUTER PROCESSING

The procedures employed to implement the AMA technique on the Univac 1108 were based on performing simulation and malfunction analysis only for a specific no-go test point encountered while running a test procedure. The procedures utilized successfully in a previous application on the IBM 7094 were based on performing all required simulation and malfunction analysis for the entire test procedure prior to conducting the actual test procedure.

The AMA program normally requires from 2 to 3 minutes of CPU time to process a typical no-go discrete indicator pattern. The average CPU time based on a representative group of nine separate runs was 2.48 minutes. However, a comparison of the processing time required for the 1108 and 7094 may be made based on the total processing time required to achieve the end result; i.e., display of the malfunctions causing a particular no-go. The figures from the 7094 experience have been adjusted to reflect the test procedure used with the 1108. This is shown in Table 2. The preprocessing of the data for the Univac takes only 16 minutes while the total preprocessing and analysis would take 105 minutes on the 7094. The on-line simulation and analysis with the 1108 requires 2.5 minutes. Since all analysis on the 7094 was pre-test, only a tape search was required, approximately 1 minute.

The AMA program file is a segmented absolute program occupying approximately 12 contiguous fastrand tracks. When executed, the AMA program utilizes approximately 45 000 core locations, consisting of 9000 program locations and 36 000 data locations. The sizing for the block of data locations was based on data processing requirements for a system model of approximately 3200 variables. AMA is a tool rather than an end item; therefore, its value depends upon how it is applied and the value of the information produced. The AMA technique can be applied in two different modes of operation for future space programs support: (1) onboard, depending upon the size and capability of the data processing system and the total operational system requirements; and (2) as a ground based analytical technique with the inputs and outputs relayed to a space station by the telemetry systems. The application of AMA can be discussed independently of the exact mode of application. The following are some typical examples of how the AMA technology could be utilized to improve the operational capability of a space system.

Onboard Maintenance

In its most direct application, AMA answers the following question. Given a complex system with a set of system indicators, the AMA programs will perform a rigorous analysis to say: Given a subset of system indicators that indicate a no-go condition at a particular point in the operational sequence, the following are the only components that would cause the specific no-go indicator pattern to occur if they failed. To restate in simpler terms, AMA can say to a space crew, for any possible failure pattern, replace these components to correct the malfunction. Depending upon the data processing capabilities of the space system, this type of data could be generated onboard using the AMA programs or the information generated before a flight on the ground and stored in a data retrieval file. It could also be transmitted to the onboard system by the telemetry system.

System Monitors

To be able to generate complete malfunction information as described in the previous paragraph, implies that a sufficient number of system monitoring points have been designed into the system. If the AMA programs are used during the design phase, they can insure that a total monitoring capability has been incorporated into the system using the minimum number of test points for the monitoring system. It can also insure that there are no

TABLE 2. RUN ANALYSIS

MODEL COMPARISON	1108	7094
DEMONSTRATION MODEL SIZE		
VARIABLES		
Discrete Indicators	224	130
Other Active Variables	2308	2295
Inactive Variables	523	624
Total	3055	3049
EQUATIONS	2265	2413
PROCESSING FOR TP321	1108 Time	7094 Time
Model Processing	16 min.	18 min.
Simulation	-	7 min.
AMA & AMAEDIT	-	80 min.
SUMMARY		
Preprocessing and file generation and storage prior to utilization for test	16 min.	105 min.
Retrieval, analysis, and display	2.5 min.	1.0 min. *

*This is for a tape search only, no analysis.

components in the final system that could fail and not be detected by one of the system indicators.

Procedure Validation

AMA requires the preparation of a system model and the use of discrete network simulation program. One of the inputs used by the simulation program is the normal test or operational procedures that provide the driving function for the system simulation. This system model can be used to either validate or assist in the development of the test or operational procedures. If the procedures are prepared by a third party, the simulation program can be used to determine the accuracy of the procedures.

System Engineering

The development and validation of the logic model required with the DNS programs is an effective method for insuring the development of effective system specialists. The results of the simulation depict the normal operational sequence of a complex system with a minimum of manual analysis of the schematic drawings. The simulation

provides a means of investigating the effects of proposed design changes and it provides a rigorous printed output that can be used to establish positive control over system interfaces.

Cost Effectiveness

There are other areas where DNS/AMA could be effectively applied, such as to determine what modes of degraded operation are possible with a component failure and to reduce the amount of ground maintenance testing. These applications require additional study and development.

MARSYAS—A SOFTWARE SYSTEM FOR THE DIGITAL SIMULATION OF PHYSICAL SYSTEMS

By

H. Trauboth and N. Prasad*

SUMMARY

In the design analysis, evaluation, and check-out of complex aerospace systems, many simulations of varying depth have to be performed to test all possible mission conditions. The Marshall System for Aerospace Systems Simulation (MARSYAS) is a software system that allows easy setup and control of the simulation of the continuous and discrete dynamics of large physical systems on a digital computer. It is particularly suited to the engineer who has little experience in simulation and computer programming. The physical systems are modeled in the form of block diagrams. The blocks can have multiple inputs and multiple outputs, and a block can contain other blocks within itself; i.e., block diagrams can be built in hierarchies. The block diagram can contain analog computer elements, transfer functions, algebraic equations, and logical functions. Elements of the block diagram that are used frequently are available in a Standard Elements List. This list is not fixed; it can be updated easily. For infrequently used elements, a FORTRAN subroutine can be submitted. MARSYAS should also serve as a storage and retrieval system for models as a basis for a "model configuration control" system on the central time-shared computer, UNIVAC 1108. The development of models is costly, and, therefore, they should be utilized by as many people as possible. The language allows a standard description of models and easy modification of already stored models specifying their blocks or hierarchies of blocks and their interconnections with names as in engineering drawings. The outputs of the simulation system can be not only time-responses but also other analysis data such as stability parameters, steady-state response, frequency response, power spectrum, and sensitivity.

The models referred to in this paper can be mathematically described by ordinary differential equations, algebraic equations, and logical functions. To provide the additional analysis capability and to obtain efficient computation speed, analysis techniques of modern control theory have been employed for the mathematical foundation of the simulation system. The differential equations generated from the block diagram are in the form of vector-matrix state equations, which do not need to be ordered for their numerical integration. Several integration modes can override the standard mode. Algebraic loops are identified by the processor. The mathematical foundation has great potential for expanding the capability and improving the computational efficiency of MARSYAS. For instance, during the numerical solution cycle, matrix multiplications and additions have to be performed that could be done simultaneously on several parallel processors and thereby speed up the simulation tremendously.

The language is designed in such a way that the user has to transmit to the computer only such information that is essential to describe the model and to specify the simulation run, but the user does not concern himself with the programming of the computer. If an engineer has no knowledge of modeling, he can call up a pre-stored model and run a simulation after specifying only the model input signals. On the other hand, if the engineer has FORTRAN programming knowledge, the language gives him some programming capability for special control of the simulation. The MARSYAS language is divided into modules that describe independent functions of the simulation. Within these modules, several MARSYAS statements are available that are written in free format and need no special ordering.

The MARSYAS software compiles a MARSYAS program, which describes a model and specifies

* The authors wish to extend special credit to Mr. H. Wisnia, Mr. H. Gabow, and Mr. H. Smith of Computer Applications, Incorporated, who made major contributions in the language implementation and software design, and to Dr. R. Sevigny and Mr. T. Balentine of Computer Sciences Corporation, who are responsible for the software implementation and programming.

the simulation, into a FORTRAN program that contains the arrays and subroutines for the numerical solution of the various matrix equations. It is built for time-sharing operations on MSFC's central computing facility, UNIVAC 1108. The software allows several users access to MARSYAS and its models simultaneously from remote stations. Although maximum use is made of the UNIVAC 1108-EXEC VIII operating system, the MARSYAS software could be used at other facilities since it is written in FORTRAN. To have the MARSYAS system evolve while it is in operation, it is mandatory to break it into many relatively independent program modules and tables.

The ease and speed of setting up and running a precisely repeatable simulation together with its special analysis capability make MARSYAS a valuable tool for analyzing and evaluating complex aerospace systems.

INTRODUCTION

New aerospace systems that will be developed in the next decade such as the Space Shuttle Vehicle and Space Station will be used for more versatile missions; they will be more autonomous and independent from ground operations; and, therefore, their design will be more complex than that of present space flight systems. To design these new systems optimally for all mission phases, extensive design analyses, evaluations, and tradeoff studies have to be performed before a design can be finalized. This means that many simulations of varying depth have to be run to test all possible mission conditions. Then, the integrated hardware and software systems have to undergo extensive testing and checkout before they are flight ready.

Several years ago, the Quality Assurance & Reliability Laboratory conducted a feasibility study to determine if simulation of large physical systems on a digital computer would aid the checkout engineer [1]. The study showed that digital simulation would indeed be a very desirable tool that would enable the checkout engineer to design and evaluate test procedures prior to hardware delivery. It would also assist him in determining critical test points and provide a better insight into the functions of the equipment that must be checked out. The Computation Laboratory then began designing the simulation system more than two years ago.

For the Apollo Program, several enormous simulation facilities have been installed at contractor and NASA sites. These consist of various types of simulators such as special purpose hardware simulators; flight trainers; and analog, hybrid, and digital computers [2-4]. Analog computers and special purpose simulators have played a major role in simulation until a few years ago, although first attempts were made to simulate analog computer block diagrams on a digital computer as early as 1955 [5]. Since then, the great flexibility of modern digital computers has been explored in a number of developments of digital simulation languages particularly for non-realtime analyses [6]. To direct the development of digital simulation languages, a standard language was introduced that is particularly suited for the simulation of analog computer-like block diagrams mixed with FORTRAN subroutines and statements [7]. Most of the common digital simulators are precompilers that generate FORTRAN code and order the integrator statements automatically so that the numerical integration for each integrator can be performed in the proper sequence. In comparison, the digital simulation system described in Reference 8 interprets linear block diagrams of transfer functions and converts them into a matrix equation whose coefficients are determined by the numerical convolution for each transfer function block [9].

A blueprint of the digital simulation system described in this paper was given in Reference 10. This simulation system addresses itself to the engineer who has little experience in simulation and in computer programming and who wants to simulate large physical systems. It should be used for a variety of applications such as for design analysis and evaluation, checkout, and malfunction analysis. This simulation system should also serve as a storage and retrieval system for models as a basis for a "model configuration control" system on a central time-shared computer. The development of models is costly, and, therefore, they should be utilized by as many people as possible. The language allows a standard description of models and easy modification of already stored models, assuming the physical system is described by multiple input/output blocks or hierarchies of blocks and their interconnections using names as they appear in engineering drawings. The outputs of the simulation system are not only time-responses but also other analysis data such as stability parameters, steady-state response, frequency response, power spectrum, and sensitivity.

During the operation of a system of this magnitude, one has to reckon with certain alterations of the operational features of the system. Therefore, the simulation systems software is designed in a modular form to keep the impact of possible design changes to a minimum. To provide the additional analysis capability and to obtain efficient computation speed, analysis techniques of modern control theory have been employed for the mathematical foundation of the simulation system. The differential equations generated from the block diagram are in the form of vector-matrix state equations, which do not need to be ordered for their numerical integration.

The description of the user language, the mathematical foundations, and the software structure will be brief in this paper; however, separate papers are being prepared to cover these areas more exhaustively.

SIMULATION CAPABILITY

Before a physical system can be simulated, a model of its functions has to be generated. In most cases, the derivation of a model requires human judgement and, therefore, is a manual process. Only in special cases, such as electrical circuits, does a direct relationship exist between the physical components network and the mathematical description of its functions. In such a case the physical network can be described directly to the computer which then generates the mathematical model and solves for it [11, 12]. For the design of MARSYAS, a model of the guidance and control system of the Saturn V and a model of the propulsion system of the S-IVB stage were used as test models representative of other space vehicles' electrical, mechanical, and hydraulic systems. The models referred to in this paper represent the continuous and discrete dynamics of physical systems which can be mathematically described by ordinary differential equations, algebraic equations, and logical functions.

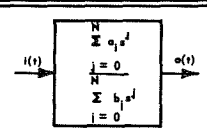
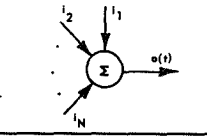
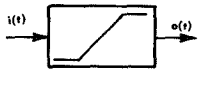
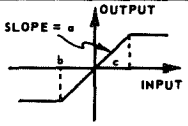
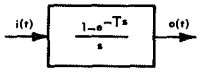
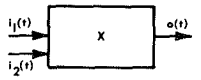
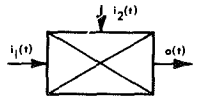
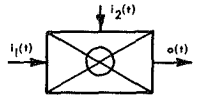
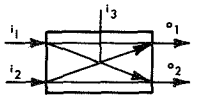
The engineer prefers to describe a model by block diagrams because their graphical representation is visually comprehensive. The blocks of the diagram can have multiple inputs and multiple outputs, and a block can contain other blocks within itself; i.e., block diagrams can be built in hierarchies, or in other words they can be nested at several levels. At the lowest level where the block cannot be broken down further, the block is called an element. There are linear and nonlinear elements. A linear element is represented by a transfer function or more generally by a linear differential equation. A nonlinear element is represented by an algebraic equation or by a logical or switching function or by a nonlinear differential equation. Elements that are used frequently are called standard elements and are available in a Standard Elements List (Table 1). This list is not fixed; it can be updated easily using MARSYAS statements. For infrequently used special elements, a FORTRAN subroutine can be submitted. Thus, the block diagram can contain analog computer elements, transfer functions, algebraic equations, and nonlinear ordinary differential equations. Figure 1 depicts a typical block diagram of a model.

A block diagram is specified to the computer only by the names of the blocks, inputs, and outputs; by the names and values of the element parameters; and by the unidirectional interconnections of the block. The names can be up to 36 characters long, so that the same names as found in engineering documentation can be used. The block diagram can be stored permanently in the Functional Data Base (FDB) or an already-stored model can be modified. Only authorized personnel having the access-key can write into the FDB, whereas everybody can read out and use models of the FDB.

For a simulation run, the input signals or excitation functions can be pre-stored analytical functions such as exponential sinusoidal time functions or digitized signals recorded on magnetic tape. These recorded signals may be measured signals or output signals generated by a previous simulation run. The outputs of the simulation can be manifold. Any connnection point in the block diagram can be chosen for obtaining a systems output signal. The dynamic systems output signals can be plotted or printed as functions of time. The steady-state response can be calculated from the matrices available. For linear systems, additional analysis information can be computed such as frequency response, power spectrum, stability, and parameter sensitivity.

The ability for the user to control the internal processing of the various simulation functions is kept to a minimum and restricted to functions essential to the simulation. The user can specify the relative truncation error, the integration step

TABLE 1. EXTRACT FROM LIST OF STANDARD ELEMENTS

CLASS	BLOCK DIAGRAM SYMBOL	# OF INPUTS	# OF OUTPUTS	MNE-MONIC	INPUT – OUTPUT RELATION	LIST OF PARAMETERS IN THE ORDER IN WHICH THEY APPEAR IN THE ELEMENTS STATEMENT
BLOCK		1	1	BL	$\sum_{i=0}^{N} b_i \frac{d^i o(t)}{dt^i} = \sum_{i=0}^{N} a_i \frac{d^i i(t)}{dt^i}$	$N, a_N, a_{N-1}, a_{N-2}, \ldots, a_o, b_N, b_{N-1}, b_{N-2}, \ldots, b_o$
CONSTANT MULTIPLIER		1	1	CM	$p(t) = K\, i(t)$	K
ADDER		N	1	AD	$o(t) = \sum_{i=1}^{N} i_i(t)$	NONE
LIMITER		1	1	LM		a, b, c
SAMPLE AND HOLD		1	1	SH	$o(t) = i(nT),\ nT \le t < (n+1)T,$ $n = 0, 1, 2, \ldots$	T
MULTIPLIER		2	1	ML	$o(t) = i_1(t) \cdot i_2(t)$	NONE
BOOLEAN RELAY		2	1	BR0	$o(t) = \begin{cases} i_1(t), & i_2(t) \neq 0 \\ 0, & i_2(t) = 0 \end{cases}$	NONE
		2	1	BR1	$o(t) = \begin{cases} 0, & i_2 \neq 0 \\ i_1(t), & i_2 = 0 \end{cases}$	NONE
RESOLVER		3	2	RE	$\begin{bmatrix} o_1 \\ o_2 \end{bmatrix} = \begin{bmatrix} \cos(i_3) & \sin(i_3) \\ -\sin(i_3) & \cos(i_3) \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix}$	NONE

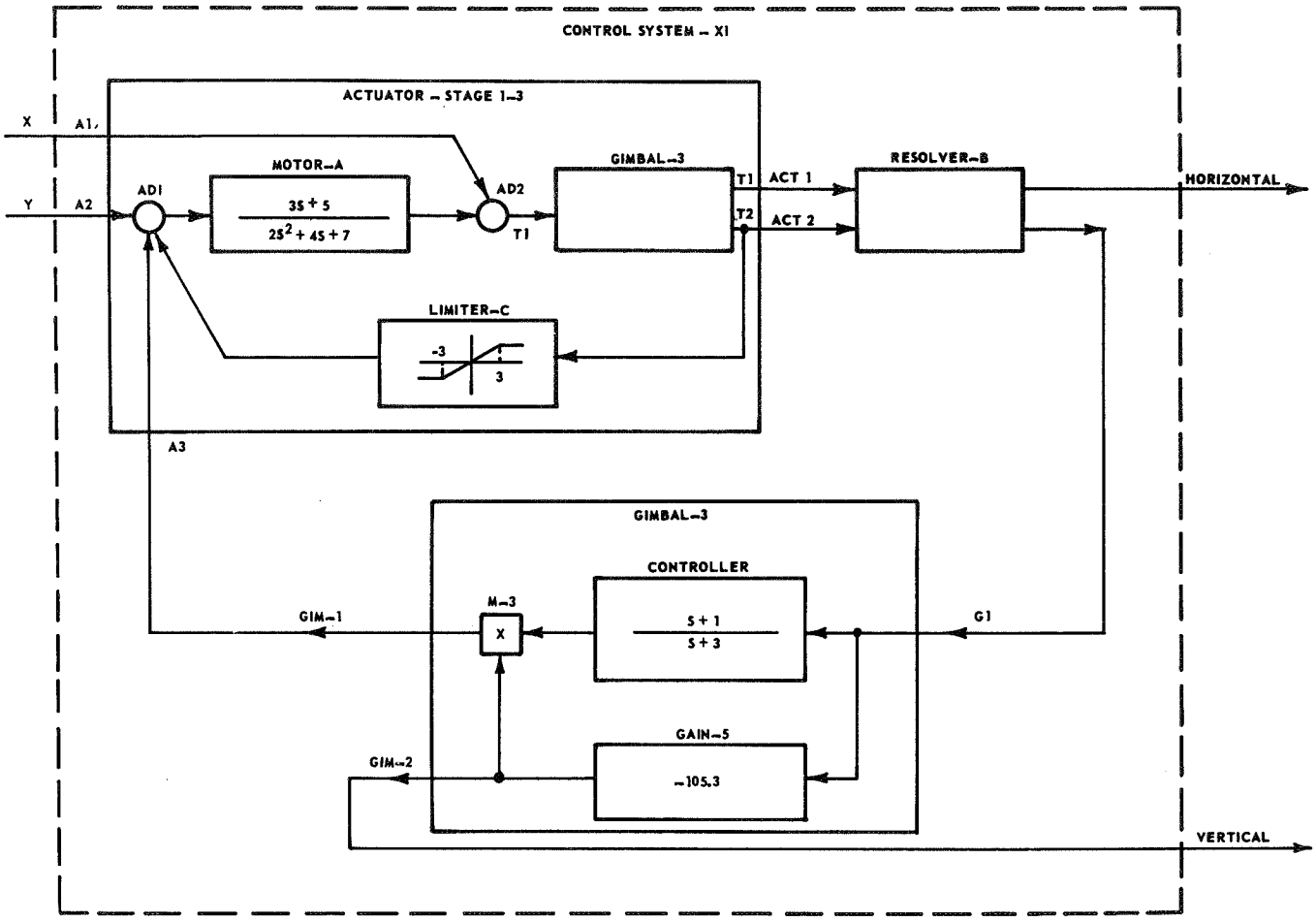


FIG. 1 EXAMPLE OF A MODEL DESCRIBED BY A BLOCK DIAGRAM OF MULTIPLE INPUT/ OUTPUT BLOCKS AND A NESTED BLOCK.

Figure 1. Example of a model described by a block diagram of multiple input/output blocks and a nested block.

size, or the numerical integration method if he wants to override the standard built-in method. Algebraic loops are identified automatically by the software.

ENGINEER-ORIENTED LANGUAGE

The language is designed in such a way that the user has to transmit to the computer only that information which is essential to describe the model and to specify the simulation run, but does not concern himself with the programming of the computer. If an engineer has no knowledge of modeling he can call up a pre-stored model and run a simulation after specifying only the model input signals. On the other hand, if the engineer has FORTRAN programming knowledge, the language gives him some programming capability for special control of the simulation.

The MARSYAS language is divided into modules that describe independent functions of the simulation. These language modules are the Description Module, Modification Module, Simulation Module, Continuation Module, Post-Processing Module, and Analysis Module. Within these modules, MARSYAS statements are written in free format and need no special ordering. However, the ordering of the modules themselves has to follow a simple logical rule; e. g., a Simulation Module has to be preceded by a Description Module because the model must first be described before it can be simulated. A statement consists of an operator and an argument, which can be subdivided into several variable argument fields depending on the "operator."

The Description Module is used to describe the model given in form of a block diagram. It is headed by the MODEL-statement. The ELEMENTS-statement contains the name of the element, the type of standard

element (to be found in the List of Standard Elements), and the parameters. The parameters are written in the proper format for the particular element type and are either the numerical values or names. The numerical value of a named parameter is given by the PARAMETER-statement. If the element is not in the Standard Element List, a FORTRAN subroutine carrying the same name as the element follows the ELEMENT-statement. A block that is stored in the Functional Data Base (FDB) is called by the SUBMODEL-statement containing the embedded model name and its input and output names. The CONNECT-statement carries the names of strings of inputs and outputs of elements, submodels, systems inputs, or systems outputs to be connected to form the model block diagram. For elements or submodels having a single input/output, only the name of the element or submodel appears in the CONNECT-statement. The INPUT-statement designates names to the inputs of the model, and the OUTPUT-statement designates names to the outputs of the model. The STORE-statement, carrying the proper key-code, transfers the Description Module into the permanent FDB. Thus, the Description Module can be used for storing and retrieving models as well as for describing the model for a subsequent simulation run. Figure 2 illustrates the MARSYAS program for the example shown in Figure 1.

The Modification Module allows inserting, deleting, and disconnecting of elements and submodels through the use of the INSERT- and DISCONNECT-statements. Statements of the Description Module are used to specify the elements, parameters, and interconnections that are to be modified. The Modification Module can be used for modifying models of the FDB or for a subsequent simulation.

The Simulation Module is used to define the course of the simulation. The INITS-statement specifies non-zero initial conditions at the outputs of the linear elements and submodels. The EXCITE-statement tells the MARSYAS processor what excitation functions or record tapes are fed into what systems inputs of the model. If a numerical integration method other than the standard method is to be used, the INTMOD-statement specifies the integration method and the relative truncation error or the integration step-size (for fixed step-size integration methods). The IF-statement determines the condition under which the simulation should terminate or hold; e.g., if a certain time or certain amplitudes of certain output signals have been reached. If the simulation calls for the repetition of a simulation run with modified parameters, the VARY-statement specifies the parameters to be changed and their values. The STOP-statement terminates, and the HOLD-statement discontinues the simulation.

The Continuation Module initiates the temporary storing of intermediate results that are necessary to continue the simulation run at a later time. This module is particularly useful when the user wishes to insert check points to obtain intermediate outputs in a lengthy simulation run. (The user can also change the integration mode at these check points.) Based on these outputs the user can decide if the run is worth continuing.

In the Post-Processing Module, the user indicates which output signals he wishes to print or plot and the format and labels of the output. The FORMAT-statement resembles the FORMAT-statement in FORTRAN.

The Analysis Module allows the user to designate the type of analysis output he wishes; e.g., the FREQ-statement calls for the frequency response within the specified frequency range. Other analysis information includes steady-state response, power spectrum, stability, and sensitivity for which special statements are available.

MATHEMATICAL FOUNDATION

Analytical Formulation

In the formulation of the mathematical process that converts the block diagram into an internal format digestible by the computer, we distinguish between three parts of the model: (1) the "dynamic" elements, (2) the "non-dynamic" elements, and (3) their interconnections. A "dynamic" element has the property of storing signal information while a "non-dynamic" element responds instantaneously to an input signal. The 'constant multiplier' (or ideal amplifier) and the 'summer' are linear "non-dynamic" elements. The linear elements 'time-delay,' 'sample-and-hold,' and 'differentiator' are treated as pseudo-nonlinear elements. For explaining the mathematics, it is assumed that the model consists of interconnected "dynamic" and "non-dynamic" elements of various types but of no

```
BEGIN, MARSYAS - PROGRAM - X $
  MODEL, ACTUATOR - STAGE 1-3 $
    INPUTS, A1, A2, A3 $  OUTPUTS, ACT1, ACT2 $
    ELEMENTS, BL, MOTOR -A (2,0,3,5,2,4,7) $
        = , LM (1, -3,3) $
  SUBMODEL, GIMBAL-3 (IN, G1) (OUT, GIM-1, GIM-2) $
    NAMING, G1, I1, GIM-1, T1, GIM-2, T2 $
    ELEMENTS, AD, AD1, AD2 $
  CONNECT, A2, AD1, MOTOR-A, AD2, I1, T2, LIMITER-C,
        AD1 $  =, A1, AD2 $  =, A3, AD1 $
  =, T1, ACT1, T2, ACT2 $
  END $
MODEL, CONTROL SYSTEM - X1 $
  INPUTS, X, Y $        OUTPUTS, HORIZONTAL, VERTICAL $
  ELEMENTS, RE, RESOLVER -B      $
  SUBMODEL, ACTUATOR-STAGE 1-3 (IN, A1, A2) (OUT, ACT1, ACT2) $
        =, GIMBAL-3 (IN, G1) (OUT, GIM-1, GIM-2) $
  CONNECT, X, A1, ACT1, RESOLVER-B (U1), RESOLVER-B (W1), HORIZONTAL $
  =, Y, A2, ACT2, RESOLVER-B(U2), RESOLVER-B(W2),
    G1, GIM-1, A3 $  =, GIM-2, VERTICAL $
  END $
SIMULATE, CONTROL SYSTEM - X1 $
  INITS, MOTOR - A (1.5, 12) $
  EXCITE, FSTEP (5.0), X, FSIN (1, 3000, 0), Y $
  IF, TIME.GT.2.0, STOP $
  PRINT-STEP, 0.01, X, Y, HORIZONTAL, VERTICAL $
  END $
END. MARSYAS - PROGRAM - X $
```

Figure 2. MARSYAS — Program of the example shown in Figure 1. (It is assumed that model GIMBAL-3 is stored in the functional data base.)

nested submodels. By some software processing, the MARSYAS processor has already unwrapped these nested submodels.

The linear "dynamic" element 'transfer function' is characterized by the following relationship:

$$\sum_{k=0}^{p} b_k \frac{d^k o(t)}{dt^k} = \sum_{k=0}^{q} a_k \frac{d^k i(t)}{dt^k} \quad , \qquad (1)$$

where a_k and b_k are constant coefficients and p is the order of the differential equation (or number of poles in the complex frequency domain). It is assumed that $q < p$. If $p = q$, the 'transfer function' element can simply be split into one with $q < p$ and one 'constant multiplier' and 'summer' element. The output signal is $o(t)$ and $i(t)$ is the input signal of the element. Using the method as described in References 13 and 14, this differential equation can be converted into a state variable matrix equation. For the jth element the following is obtained:

$$\dot{X}^{(j)}(t) = A^{(j)} X^{(j)}(t) + P^{(j)} i^{(j)}(t) \qquad (2a)$$

$$o^{(j)}(t) = C^{(j)} x_1^{(j)}(t) \quad . \qquad (2b)$$

$$X^{(j)}(t) = \begin{vmatrix} x_1^{(j)} \\ \cdot \\ \cdot \\ \cdot \\ x_p^{(j)} \end{vmatrix}$$ is a state vector of the jth element, and $\dot{X}^{(j)}(t)$ its time derivative. $A^{(j)}$, $C^{(j)}$, and $P^{(j)}$ are constant real matrices of the dimension $p \times p$ and can be obtained by algebraic calculations from a_k and b_k of equation (1):

$$A^{(j)} = \begin{vmatrix} -\frac{b_{p-1}}{b_p} & 1 & 0 \ldots 0 \\ -\frac{b_{p-2}}{b_p} & 0 & 1 \ldots 0 \\ \\ -\frac{b_1}{b_p} & 0 & 0 \ldots 1 \\ -\frac{b_0}{b_p} & 0 & 0 \ldots 0 \end{vmatrix} \qquad P^{(j)} = \frac{1}{b_p} \begin{vmatrix} a_{p-1} \\ a_{p-2} \\ \cdot \\ \cdot \\ a_0 \end{vmatrix} \qquad C^{(j)} = \begin{vmatrix} 1 & 0 \ldots 0 \\ \text{---------} \\ 0 & 0 \ldots 0 \end{vmatrix} \qquad (2c)$$

For a collection of m "dynamic" elements, the following can be written:

$$\dot{X}(t) = AX(t) + PI(t) \qquad (3a)$$

$$O(t) = CX(t) \quad , \qquad (3b)$$

where

$$\dot{X}(t) = \begin{vmatrix} \dot{X}^{(1)}(t) \\ \cdot \\ \cdot \\ \dot{X}^{(j)}(t) \\ \cdot \\ \cdot \\ \dot{X}^{(m)}(t) \end{vmatrix} \quad , \quad X(t) = \begin{vmatrix} X^{(1)}(t) \\ \cdot \\ \cdot \\ X^{(j)}(t) \\ \cdot \\ \cdot \\ X^{(m)}(t) \end{vmatrix} \quad ,$$

$$A = \begin{vmatrix} A^{(1)} & & & & \\ & \cdot & & & \\ & & \cdot & & \\ & & & A^{(j)} & \\ & & & & \cdot \\ & & & & & \cdot \\ & & & & & & A^{(m)} \end{vmatrix} \quad ,$$

$$P = \begin{vmatrix} P^{(1)} & & & & \\ & \cdot & & & \\ & & \cdot & & \\ & & & P^{(j)} & \\ & & & & \cdot \\ & & & & & \cdot \\ & & & & & & P^{(m)} \end{vmatrix} \quad ,$$

$$C = \begin{vmatrix} C^{(1)} & & & & \\ & \cdot & & & \\ & & \cdot & & \\ & & & C^{(j)} & \\ & & & & \cdot \\ & & & & & \cdot \\ & & & & & & C^{(m)} \end{vmatrix} \quad ,$$

$$I(t) = \begin{vmatrix} i^{(1)}(t) \\ \cdot \\ \cdot \\ i^{(j)}(t) \\ \cdot \\ \cdot \\ i^{(m)}(t) \end{vmatrix} \quad , \text{ and } O(t) = \begin{vmatrix} o^{(1)}(t) \\ \cdot \\ \cdot \\ o^{(j)}(t) \\ \cdot \\ \cdot \\ o^{(m)}(t) \end{vmatrix} \quad .$$

It is now assumed that the "dynamic" elements are connected in any way through 'constant multipliers' and 'summers' to form a linear model. The following linear interconnection matrix equations can then be written:

$$I(t) = EO(t) + FU(t) \tag{4a}$$

$$W(t) = GO(t) + HU(t) \quad , \tag{4b}$$

with

$$E = \begin{vmatrix} e_{11} & e_{12} \cdots e_{1m} \\ e_{21} & e_{22} \cdots e_{2m} \\ & \vdots \\ e_{m1} & e_{m2} \cdots e_{mm} \end{vmatrix} \quad ,$$

$$F = \begin{vmatrix} f_{11} & f_{12} \cdots f_{1k} \\ f_{21} & f_{22} \cdots f_{2k} \\ & \vdots \\ f_{m1} & f_{m2} \cdots f_{mk} \end{vmatrix} \quad ,$$

$$G = \begin{vmatrix} g_{11} & g_{12} \cdots g_{1m} \\ g_{21} & g_{22} \cdots g_{2m} \\ & \vdots \\ g_{l1} & g_{l2} \cdots g_{lm} \end{vmatrix} \quad ,$$

and

$$H = \begin{vmatrix} h_{11} & h_{12} \cdots h_{1k} \\ h_{21} & h_{22} \cdots h_{2k} \\ & \vdots \\ h_{l1} & h_{l2} \cdots h_{lk} \end{vmatrix} \quad ,$$

where

$U(t)$ = vector of inputs (excitations) into model

$W(t)$ = vector of outputs from model

m = number of "dynamic" elements in model

k = number of inputs into model

and

l = number of outputs from model.

The coefficient e_{ij} in E means the total constant gain along the path from the output o_j of "dynamic" element j to the input i_i of "dynamic" element i. The coefficient f_{ij} in F is the total constant gain along the path from the model input u_j to the input o_i. The coefficient g_{ij} in G is the total constant gain from the output o_j to the model output w_i. The coefficient h_{ij} in H is the total constant gain from the model input u_j to model output w_i.

By substituting equation (4a) into equation (3a) and equation (3b) into equation (4b), the model overall matrix equations are obtained:

$$\dot{X}(t) = A^*X(t) + P^*U(t) \tag{5a}$$

and

$$W(t) = C^*X(t) + D^*U(t) \quad , \tag{5b}$$

where

$$A^* = A + PEC, \quad P^* = PF$$

$$C^* = GC, \text{ and } \quad D^* = H \quad . \tag{5c}$$

Nonlinear elements of the form

$$y^{(j)}(t) = f^{(j)}(r^{(j)}(t),\ t) \qquad (6a)$$

are now included, where $y^{(j)}(t)$ denotes the output vector, $r^{(j)}(t)$ the input vector, and $f^{(j)}$ the function of the jth nonlinear element. For a collection of nonlinear elements it is

$$Y(t) = \overline{F}\left[R(t)\right] \quad . \qquad (6b)$$

For the inputs to all "dynamic" elements and to all nonlinear elements, respectively, the following nonlinear interconnection matrix equations are written:

$$I(t) = E\,O(t) + F\,U(t) + K\,Y(t) \qquad (7a)$$

and

$$R(t) = E'O(t) + F'U(t) + K'Y(t) \quad , \qquad (7b)$$

where the vectors $Y(t)$ and $R(t)$ represent the collection of the output vectors and input vectors of all nonlinear elements of the model. The output vector $W(t)$ for the model becomes

$$W(t) = G\,O(t) + H\,U(t) + K''Y(t) \quad . \qquad (7c)$$

The matrices E . . . E", F . . F", and K . . . K" represent the cumulative gain along the various paths between the inputs and outputs of the "dynamic" elements, nonlinear elements, and the model.

By substituting equation (7a) into equation (3a) and equation (3b) into equation (7c) and using equations (6a) and (7b), the following are obtained:

$$\dot{X}(t) = A^*X(t) + P^*U(t) + N(O, U, T) \qquad (8a)$$

and

$$W(t) = C^*X(t) + D^*U(t) + M(O, U, t) \quad , \qquad (8b)$$

with A^*, P^*, C^*, and D^* being the same matrices as in equation (5c). $N(O, U, t)$ and $M(O, U, t)$ are the nonlinear column vectors $[n_1(O, U, t),\ n_2(O, U, t),\ \ldots\ n_m(O, U, t)]$ and $[m_1(O, U, t),\ m_2(O, U, t),\ \ldots\ m_l(O, U, t)]$ respectively. It is assumed here that there are no "algebraic loops" in the model. An overview diagram of the mathematical process is given in Figure 3.

The matrices A^*, P^*, C^*, and D^* are characteristics for a linear model and can be used for a number of analyses. The stability can be obtained from the eigenvalues of matrix A^*; i.e., the model is unstable if there is at least one eigenvalue in the right half of the complex frequency plane. The frequency response can be computed in the following way. The analytical solution of equation [5a] is

$$X(t) = e^{A^*t}X(0) + \int_0^t e^{A^*(t-\lambda)}\,P^*U(\lambda)\,d\lambda \quad . \qquad (9)$$

By applying the impulse function I at the model inputs, the impulse responses or weighting functions at the model outputs are obtained, so that for $U(t) = I$ (= identity matrix) and $X(0) = 0$ equation (9) becomes

$$X_o(t) = e^{A^*t}P^*I \quad . \qquad (10)$$

By applying the Fourier transformation to $X_o(t)$, $X_o(jw)$ is obtained and from equation (5b)

$$W(jw) = C^*X_o(jw) + D^* \quad . \qquad (11)$$

The parameter sensitivity $\partial W/\partial a_{ij}$ is determined by applying $\partial X/\partial a_{ij}$ to equation (5a), where a_{ij} is a coefficient of matrix A^*. The

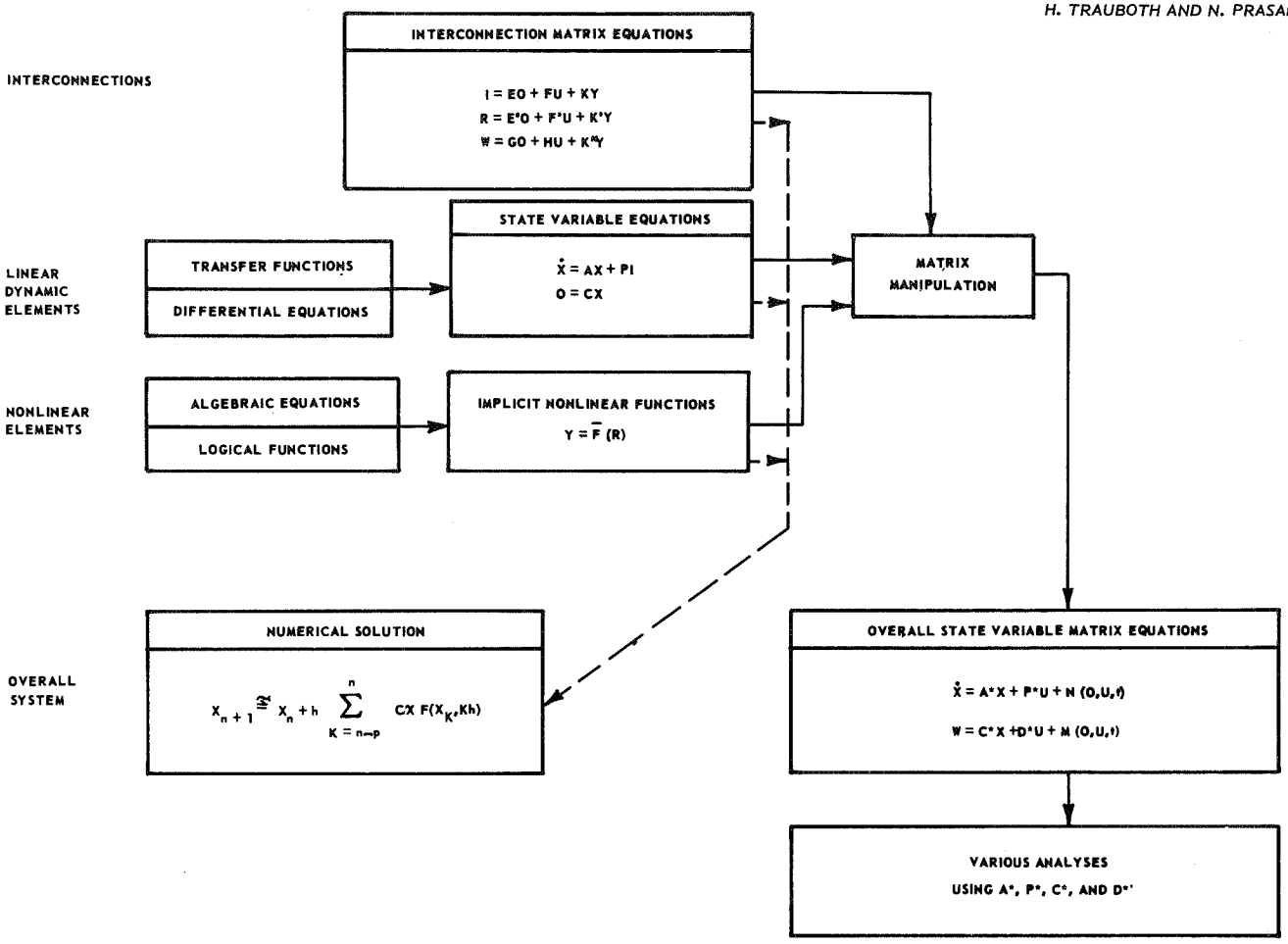

INTERCONNECTIONS
INTERCONNECTION MATRIX EQUATIONS
I = EO + FU + KY
R = E'O + F'U + K'Y
W = GO + HU + K''Y
LINEAR DYNAMIC ELEMENTS
TRANSFER FUNCTIONS
DIFFERENTIAL EQUATIONS
STATE VARIABLE EQUATIONS
Ẋ = AX + PI
O = CX
MATRIX MANIPULATION
NONLINEAR ELEMENTS
ALGEBRAIC EQUATIONS
LOGICAL FUNCTIONS
IMPLICIT NONLINEAR FUNCTIONS
Y = F̄ (R)
OVERALL SYSTEM
NUMERICAL SOLUTION
X_{n+1} ≅ X_n + h Σ_{K=n-p}^{n} CX F(X_K, Kh)
OVERALL STATE VARIABLE MATRIX EQUATIONS
Ẋ = A*X + P*U + N (O,U,t)
W = C*X + D*U + M (O,U,t)
VARIOUS ANALYSES
USING A*, P*, C*, AND D*'

Figure 3. Overview diagram of the mathematical process.

steady-state response can be found also for a nonlinear model by setting $\dot{X}(t) = 0$ in equation (8a) and obtaining

$$X(t) = -A^{*-1} P^* \left(U(t) + N(O, U, t) \right) \qquad (12a)$$

and

$$W(t) = C^* X(t) + D^* U(t) + M(O, U, t), \qquad (12b)$$

which involves the inversion of A^* prior to the numerical solution.

Numerical Solution

Nearly all numerical methods for the solution of differential equations are based on the numerical integration of first order differential equations [15-17]. Hence, the state-variable matrix equations are particularly suited for these methods. A system of first order differential equations, in general of the form

$$\dot{X}(t) = F(X(t), t) \quad , \qquad (13)$$

is usually approximated by single-step evaluation or multistep predictor-corrector methods where

$\dot{X}(t)$, $X(t)$, and $F(X(t),t)$ are the column vectors $[\dot{x}_1(t), \ldots \dot{x}_M(t)]^T$, $[x_1(t), \ldots x_M(t)]^T$; and $[f_1(X, t), \ldots f_M(X, t)]^T$, with M equal to the total number of state variables within the model. The most common single-step method is the Runge-Kutta (4th order) which approximates equation (13) for $t = (n+1)\cdot h$ into

$$X_{n+1} \cong X_n + \frac{1}{6} (K_1 + 2K_2 + 2K_3 + K_4) \qquad (14)$$

with the vectors

$$K_1 = h\, F(X_n, t_n) \quad ,$$

$$K_2 = h\, F\left(X_n + \frac{h}{2}, t_n + \frac{K_1}{2}\right) \quad ,$$

$$K_3 = h\, F\left(X_n + \frac{h}{2}, t_n + \frac{K_2}{2}\right) \quad ,$$

$$K_4 = h\, F(X_n + h, t_n + K_3) \quad ,$$

where h is the time step.

The multistep predictor-corrector methods are of the following form:

Predictor

$$X^p[(n+1)h] \cong X(nh) + h \sum_{k=n-p}^{n} c_k F[X(kh), kh] \qquad (15)$$

Corrector

$$X[(n+1)h] \cong X(nh) + h \sum_{k=n-q}^{n} d_k F[X(kh), kh] + h\, d_{n+1} F\left\{X^p[(n+1)h], (n+1)h\right\} \quad , \qquad (16)$$

with p being the order of the predictor and q being the order of the corrector polynomial, and c_k and d_k are constant coefficients depending on the method used. For instance, for the Adams-Bashforth predictor it is $c_{n-3} = -9/24$, $c_{n-2} = 37/24$, $c_{n-1} = -59/24$, and $c_n = 55/24$; and for the Adams-Moulton corrector it is $d_{n-2} = 1/24$, $d_{n-1} = -5/24$, $d_n = -19/24$, and $d_{n+1} = 9/24$ [18].

One can show that numerically solving the overall matrix equations (7) and (8) is not the most efficient way, because the matrices A^*, P^*, C^*, and D^* contain many zero elements. Less computation steps are necessary if one uses the individual equations (3), (6a), and (7).

The following is the numerical process (Fig. 4):

(1) First the excitation vector $U(t)$ is updated for $t = nh$ and the following is calculated:

(2) or

$$O(nh) = C\, X(nh) \qquad (17)$$

$$O_n = C\, X_n$$

(From now on, the subscript n will be used for the time instant $t = nh$.)

(3) Part of the input vector $R(t)$ is computed from equation (7b) for those nonlinear elements whose input is not connected to other nonlinear elements. For these nonlinear elements, the output can now be calculated using equation (6a). Then, that part of $R(t)$ can be calculated which contains known $y_n^{(j)}$. Through alternate use of equations (7b) and (6a) and assuming that the nonlinear equations were already properly ordered, the complete vector Y_n can be calculated. Now, the input vector I_n can be obtained by use of equation (7a).

(4) $$I_n = E\, O_n + F\, U_n + K\, Y_n \quad , \qquad (18)$$

and the model output vector W_n of equation (7c) is

(5) $$W_n = G\, O_n + H\, U_n + K''\, Y_n \quad . \qquad (19)$$

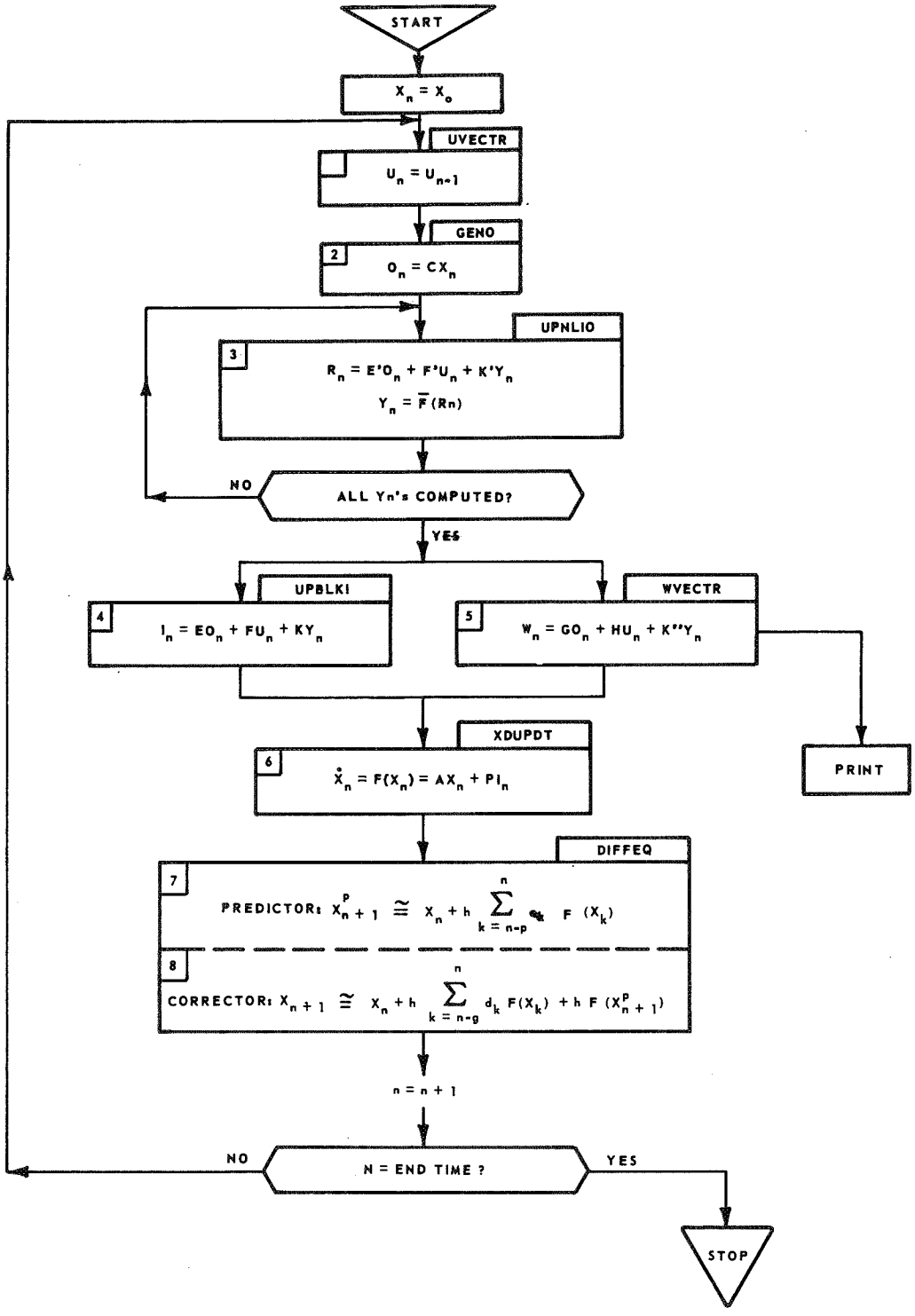


Figure 4. Flow of numerical solution of matrix equations.

Knowing I_n we go to equation (3a),

$$\dot{X}(t) = F\left(X(t), t\right) = AX(t) + PI(t) \quad ,$$

and evaluate

$$\textcircled{6}\ \dot{X}_n = F_n = AX_n + PI_n \quad . \tag{20}$$

Then, using equation (15), the predicted value of X_{n+1} is

$$\textcircled{7}\ X^P_{n+1} \cong X_n + h \sum_{k=n-p}^{n} c_k F(X_k) \quad , \tag{21}$$

and the corrected value of X_{n+1}, i.e., the final value of X, for $t = (n+1)h$ is

$$\textcircled{8}\ X_{n+1} \cong X_n + h \sum_{k=n-q}^{n} d_k F(X_k) + h F\left(X^P_{n+1}\right) \tag{22}$$

The sequence of the numerical evaluation is outlined in Figure 4. One can see that within each block, only matrix multiplications and additions have to be performed. The row-by-column multiplications are not dependent on each other; hence, the sequence in which they are executed is immaterial. This property has the advantage that several vector multiplications and additions could be computed simultaneously resulting in a tremendous speedup of the computations if several parallel processors were available.

SOFTWARE STRUCTURE

General

The prime objective of the MARSYAS software is to transform a MARSYAS program that describes a model and specifies the simulation into a FORTRAN program that contains the arrays and subroutines for the numerical solution of the various matrix equations. The MARSYAS software is, so-to-speak, a pre-compiler for compiling MARSYAS language statements into a set of FORTRAN subroutines, arrays, and control cards, i.e., the Object Program, and a controller for the execution of these FORTRAN programs. It is built for time-sharing operations on MSFC's central computing facility, UNIVAC 1108. The software should allow several users access to MARSYAS and its models simultaneously from remote stations. Although maximum use of the UNIVAC 1108-EXEC VIII operating system is made, the MARSYAS software could be used at other facilities since it is written in FORTRAN. To have the MARSYAS system evolve while it is in operation, it is mandatory to break it into many relatively independent program modules and tables. The major Program Modules (PM) are defined by the language modules. Hence, we distinguish between the Description PM, Modification PM, Simulation PM, Continuation PM, Post-Processing PM, and Analysis PM. Outside these Program Modules there are other programs, such as the FORTRAN Object Program (OP), library routines for standard elements and excitation functions, scientific subroutines for the numerical integration of first order differential equations, scanning routines, error recovery routines, and control routines. The Object Program is compiled by the FORTRAN compiler and then executed like any manually generated FORTRAN program. Figure 5 is an attempt to present an overview of the main flow of action; it is, of course, a simplification.

Object Program

The FORTRAN Object Program consists of the MAIN program which has a fixed structure and calls up various subroutines of fixed and variant structure. "Variant" means that the program length varies with each simulation run. The OP reads from four files as follows:

1. The Simulation Temporary File, which contains all arrays for solving the matrix equations of Figure 4.

2. The Change File, which contains those parameters that are to be changed as specified by the VARY-statement.

3. The Continuation Permanent File, which stores intermediate results of a discontinued simulation run such as the state vector $X(t)$.

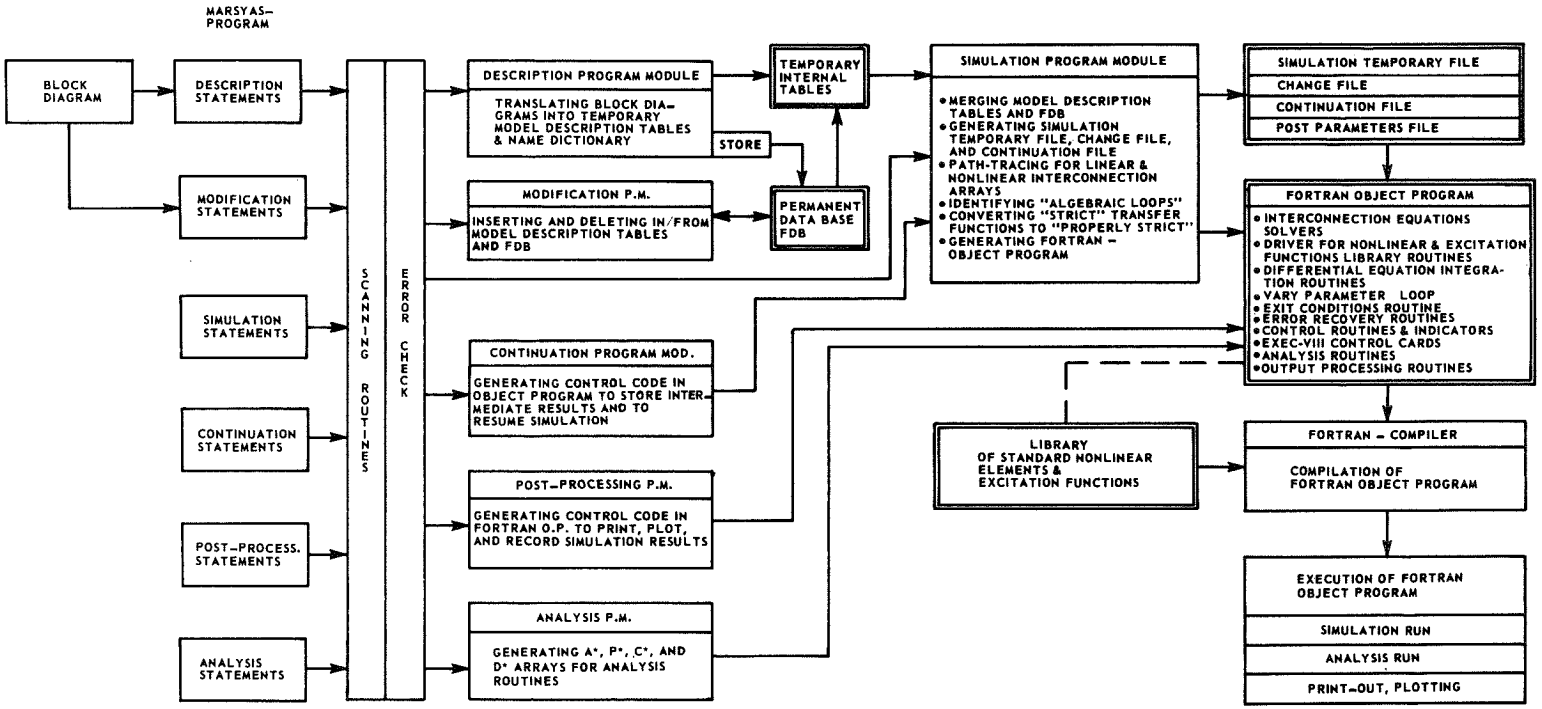

MARSYAS–
PROGRAM
BLOCK DIAGRAM
DESCRIPTION STATEMENTS
MODIFICATION STATEMENTS
SIMULATION STATEMENTS
CONTINUATION STATEMENTS
POST–PROCESS. STATEMENTS
ANALYSIS STATEMENTS
SCANNING ROUTINES
ERROR CHECK
DESCRIPTION PROGRAM MODULE
TRANSLATING BLOCK DIA–GRAMS INTO TEMPORARY MODEL DESCRIPTION TABLES & NAME DICTIONARY
STORE
TEMPORARY INTERNAL TABLES
MODIFICATION P.M.
INSERTING AND DELETING IN/FROM MODEL DESCRIPTION TABLES AND FDB
PERMANENT DATA BASE FDB
CONTINUATION PROGRAM MOD.
GENERATING CONTROL CODE IN OBJECT PROGRAM TO STORE INTER–MEDIATE RESULTS AND TO RESUME SIMULATION
POST–PROCESSING P.M.
GENERATING CONTROL CODE IN FORTRAN O.P. TO PRINT, PLOT, AND RECORD SIMULATION RESULTS
ANALYSIS P.M.
GENERATING A*, P*, C*, AND D* ARRAYS FOR ANALYSIS ROUTINES
SIMULATION PROGRAM MODULE
•MERGING MODEL DESCRIPTION TABLES AND FDB
•GENERATING SIMULATION TEMPORARY FILE, CHANGE FILE, AND CONTINUATION FILE
•PATH-TRACING FOR LINEAR & NONLINEAR INTERCONNECTION ARRAYS
•IDENTIFYING "ALGEBRAIC LOOPS"
•CONVERTING "STRICT" TRANSFER FUNCTIONS TO "PROPERLY STRICT"
•GENERATING FORTRAN – OBJECT PROGRAM
LIBRARY OF STANDARD NONLINEAR ELEMENTS & EXCITATION FUNCTIONS
SIMULATION TEMPORARY FILE
CHANGE FILE
CONTINUATION FILE
POST PARAMETERS FILE
FORTRAN OBJECT PROGRAM
•INTERCONNECTION EQUATIONS SOLVERS
•DRIVER FOR NONLINEAR & EXCITATION FUNCTIONS LIBRARY ROUTINES
•DIFFERENTIAL EQUATION INTEGRA-TION ROUTINES
•VARY PARAMETER LOOP
•EXIT CONDITIONS ROUTINE
•ERROR RECOVERY ROUTINES
•CONTROL ROUTINES & INDICATORS
•EXEC-VIII CONTROL CARDS
•ANALYSIS ROUTINES
•OUTPUT PROCESSING ROUTINES
FORTRAN – COMPILER
COMPILATION OF FORTRAN OBJECT PROGRAM
EXECUTION OF FORTRAN OBJECT PROGRAM
SIMULATION RUN
ANALYSIS RUN
PRINT–OUT, PLOTTING

Figure 5. Overview of MARSYAS systems software.

4. The Post-Parameters File, which keeps those parameters and labels that are specified in the Post-Processing Module.

The Object Program also generates the Simulation Output Tape which contains the numerical values of all model output signals and the associated time for a complete simulation run.

The subroutines of fixed structure perform the following functions:

TRNSX	transforms the initial conditions of the output signal vector $O(o)$ into the initial state vector $X(o)$.
XDUPDT	evaluates the derivatives $\dot{X}_n = F(X_n)$ of equation (20).
PCHNG	changes parameters stored in the Simulation Temporary File as specified in the Change File.
TIDY	writes intermediate simulation results into the Continuation Permanent File.
GENO	generates the output vector O_n from equation (17).

The variant subroutines are the following:

UVECTR	generates the vector U_n by accessing the proper library routines for the various excitation functions.
WVECTOR	generates the vector W_n from equation (19) using the routines UPGAIN and UPNLIO.
UPGAIN	creates the array G(I) which contains the cumulative gain of the Ith path between a "source" (variable O, U, or Y) and a "terminator" (variable I, W, or R).
UPNLIO	updates outputs of nonlinear devices, which are already sequenced in the proper order for evaluation, using library routines for the various standard nonlinear elements.
UPBLKI	generates the vector I_n from equation (18) using UPGAIN and UPNLIO.
DIFFEQ	is the general integration routine that calls a specific integration routine such as RKG for Runge-Kutta-Gill (4th order) or AM for Adams-Bashforth-Moulton predictor-corrector, etc.
POST	writes the Simulation Output File.
EDIT	prints and plots output data generated by POST.
PDP-Elements	specify the dimensions of the various COMMON-arrays in the OP routines.
ANALYSIS	is a collection of special subroutines to generate the overall matrices A^*, P^*, C^*, and D^* and to perform special analysis computations.

Description and Modification Program Module

The subroutines of the Description Module translate the MARSYAS-statements describing a model block diagram into the Model Description Tables for the elements, submodels, inputs, outputs, parameters, and connections. These tables are packed into several records and fields to save storage space. The names are converted into unique identification code words (ID) via the Name Dictionary, so that in the internal processing shorter Name ID's can be used. The connections are ordered into pairwise connections (predecessor/successor terminals). Since the statements can be written in any order, the END-routine has to check for certain formal errors such as undefined names, missing elements, and improper connections after all tables have been filled. If the model is to be stored into the permanent Functional Data Base (FDB), the temporary Model Description Tables are transcribed into the FDB-file.

The Modification Program Module subroutines are similar to those of the Description Module in the sense that they also access the Model Description Tables and modify them.

Simulation and Continuation Program Module

The subroutines of the Simulation Program Module have to perform a variety of functions.

Those tables of the Functional Data Base that contain submodels specified in the SUBMODEL statements and the temporary Model Description Tables are emerged into the Model Tables File (MTF). Blocks representing transfer functions of equal numerator and denominator order are converted into blocks where the order of the numerator is one less than the denominator. The various Simulation Module statements for initial conditions, excitations, integration mode, etc., are translated into the Simulation Specification Tables File. The Connection Tables File of the MTF is used for path tracing to generate the Gain Table that contains the cumulative gains between the various terminals. The nonlinear elements are sequenced in the Nonlinear Elements Table. From intermediate tables such as the Model Tables Files, Simulation Specifications File, etc., the final files used by the Object Program, (i.e., the Simulation Temporary File, Change File, and Continuation Permanent File) and the Program File (i.e., the Object Program) are generated. The Simulation Program Module also generates the Continuation Permanent File if a HOLD-statement is included in the MARSYAS program.

The Continuation Program Module accesses the Continuation Permanent File and places, among other control data, the X-vector into the Initial Conditions Record of the Simulation Temporary File, so that the simulation run can be continued by the Object Program.

Post-Processing and Analysis Program Module

The Post-Processing Program Module generates the Post Parameters File and the subroutines POST and EDIT of the Object Program. The Analysis Program Module generates the parameter arrays representing the matrices A^*, P^*, C^*, and D^* for the various analysis subroutines in the OP.

POTENTIALS AND IMPLEMENTATION OF MARSYAS

The scientific subroutines of the Object Program and the library subroutines have been successfully tested with linear and nonlinear test cases. The systems software is coded and is presently in the final checkout process on MSFC's time-sharing computer, UNIVAC 1108. The detailed design specifications are documented and revised [19]. The present implementation, however, does not include blocks of differential equations of arbitrary format and time-varying systems. The systems software for the various analyses has not been implemented.

The mathematical foundation and the software structure allow for expanding the capability of MARSYAS and for improving its language and internal processing. Thus, elements in the model block diagram that contain ordinary differential equations of any order in form of mathematical equations will be included. The coefficients of the differential equations can also be functions of time. While presently algebraic loops can be identified, but not solved for (except by inserting an artificial time delay or an implicit function), it is expected that a separate mathematical procedure that assures convergence can be found to solve identified algebraic loops. The matrix equation formulation of the physical system in MARSYAS resembles closely the manner in which electrical networks are mathematically described for analyzing on the digital computer [20]. Methods for partitioning sparse matrices in electrical circuit analysis might, therefore, be applicable to further improve the computation speed, particularly for large physical systems [21]. The language is structured in such a way that it should be straightforward to input block diagrams and simulation statements via graphical display into the computer and thereby enhance the man-machine communications tremendously. The MARSYAS language can still be polished to reduce the amount of writing by the user. However, language improvements will be delayed until the user has gained practical experience in the operation of MARSYAS.

As was pointed out previously, the MARSYAS language is well suited for an easy description of dynamic models, and the MARSYAS software system allows easy storage, retrieval, and modification of models in a central data bank. Thus, MARSYAS could become the basis for a "Model Configuration Control" System that keeps information concerning the functions of aerospace hardware up-to-date and available to many engineers and systems analysts in a common language, similar to the computerized Vehicle Configuration Control System that keeps the information about the configuration of the hardware up-to-date. The ease and speed of setting up and running a precisely repeatable simulation together with its special analysis capability make MARSYAS a valuable tool for analyzing and evaluating complex aerospace systems.

REFERENCES

1. Requirement specification part I and analysis of dynamic simulation methods for launch vehicle component level simulation. Apollo Support Department, General Electric Company, Daytona Beach, Florida, MSFC Contract NAS8-20060, Final Report, March 1966.

2. Gale, G. H.: The Boeing Huntsville Simulation Center. Simulation, vol. 5, no. 4, October 1965.

3. Saturn V System Development Breadboard Facility Data Plan. The Boeing Company, Document no. D5-15207, NASA Contract NAS8-5608.

4. Flight Software Development Laboratory. Document no. IBM-68-U60-0022, IBM Under NASA Contract, 1968.

5. Brennan, R. D.; and Linegarger, R. N.: A survey of digital simulation: digital analog simulator programs. Simulation, vol. 3, no. 6, December 1964.

6. Strauss, J. C.: Digital simulation of continuous dynamic systems — An overview. Proceedings of Fall Joint Computation Conference, 1968.

7. The SCi Continuous System Simulation Language (CSSL). SCi-Committee, Simulation, vol. 9, no. 6, December 1967.

8. Trauboth, H. H.: Digital simulation of general control systems. Simulation, June 1967.

9. Trauboth, H. H.: Recursive Formulas for the Evaluation of the Convolution Integral. Journal of ACM, vol. 16, no. 1, January 1969.

10. Trauboth, H. H.; Mitchell, J. R.; and Moore, J. W.: Digital simulation of an aerospace vehicle. Proceedings of ACM National Meeting, Washington, August 1967.

11. 1620 Electronic Circuit Analysis Program (ECAP). Application Program 1620-EE-02X, IBM Corporation, Data Processing Division, White Plains, N. Y., 1964.

12. NASAP 69/I, Network Analysis for Systems Applications Program. Contract NAS12-663, NASA, Electronic Research Center, January 1969.

13. Zadeh, L. A.: Linear System Theory. McGraw-Hill, New York, 1963.

14. Derusso, P. M.; et al.: State Variables for Engineers. John Wiley, New York, N. Y., 1965.

15. Benyon, P. R.: A review of numerical methods for digital simulation. Simulation, vol. 11, no. 5, November 1968.

16. Martens, H. R.: A comparative study of digital integration methods. Simulation, February 1969.

17. Henrici, P.: Discrete Variable Methods in Ordinary Differential Equations. John Wiley & Sons, New York, N. Y., 1962.

18. Conte, S. D.: Elementary Numerical Analysis. McGraw-Hill, New York, N. Y., 1965.

19. Programming Specifications for the MARSYAS System. Vol. I (1968), Vol. II and III (1969), Computer Applications, Inc., New York, N. Y., NASA Contract NAS8-21061.

REFERENCES (Concluded)

20. Branin, F. H.: Computer Methods of Network Analysis. Proceedings of the IEEE, vol. 55, no. 11, November 1967.

21. Tinney, W. F.; and Walker, J. W.: Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization. Proceedings of the IEEE, vol. 55, no. 11, November 1967.

Page intentionally left blank

REAL-TIME SPACE VEHICLE AND GROUND SUPPORT SYSTEMS SOFTWARE SIMULATOR FOR LAUNCH PROGRAMS CHECKOUT

By

H. Trauboth, C. O. Rigby, and P. Brown*

SUMMARY

A digital simulator was developed by Marshall Space Flight Center (MSFC), which simulates major functions of the Saturn V Launch Complex in real-time in response to the test programs of the two RCA-110A launch computers. The simulator consists of the SDS 930 digital computer, a specially built hardware interface unit, and a large data base containing the mathematical model of the launch vehicle and its ground support equipment (GSE). The objective of this development program was to find a new way to perform major functions of the Saturn V Breadboard with a more flexible digital computer, so that launch computer programs could be checked out, test programs could be evaluated, and the effect of malfunctions could be investigated without possible damaging effects on the expensive hardware systems under test. The launch computers send out sequences of discrete stimuli controlled by actual test programs, and the simulator responds in real-time by generating timely discrete and analog signals. The software is designed in such a way that no reprogramming is necessary when a configuration other than the Saturn V has to be simulated, as long as the hardware functions can be described by the same nomenclature for the data base. The Saturn V configuration was used as a test bed to demonstrate the flexibility of the approach. In using the simulator, the engineer communicates directly with the computer.

The vehicle and GSE function are described by large sets of logical equations that may contain analog time functions also. The logical equations can contain pickup and dropout times or thresholds resulting in time-delays; the analog functions are described by polynomials or tables. The total Saturn V is described by about 15 000 equations of varying length. The software is divided into three major areas: (1) Interface Support Software, (2) Simulation Processor, and (3) Simulator Diagnostics. The main portion is the Simulation Processor, which generates the data base in the computer from card input, evaluates the equations iteratively during the simulation run, controls many clocks, and initiates the selective printout of the simulation results. It comprises the five major phases: (1) Pre-Simulation, (2) Hold, (3) Initialization, (4) Real-Time Simulation, and (5) Post-Simulation. The design is modular and strongly influenced by many constraining factors such as limited core space, real-time responses, and disk access time. To minimize the computation time during the simulation run, as many functions as possible are assigned to the Pre-Simulation Phase prior to the Real-Time Simulation Phase.

INTRODUCTION

To assure the integrity of the flight systems, the launch of a Saturn V vehicle is preceded by a complicated chain of checkout operations, which involve a large system of checkout and launch equipment in the Launch Control Center and in the Mobile Launch Facility at the Kennedy Space Center. This checkout and launch system consists of manual checkout panels, ground support equipment, telemetry stations, data links, and two RCA-110A launch control computers. Commands initiated in the Launch Control Center are transferred by these computers to the launch vehicle under checkout.

---

* The authors wish to extend special credit to Mr. R. G. Abe of Computer Sciences Corporation who prepared the detail design and implemented the major portion of the simulation software, to Mr. T. L. Balentine and Mr. C. O. Rigby for their development of the diagnostic software, to Mr. E. E. Branstetter and Mr. R. Hull for their programming contributions, and specifically to their colleagues at Astrionics Laboratory who provided the hardware systems operations and the data base.

The computer sends out stimuli and receives responses that are evaluated based on predicted values stored in the computer memory. Sending out stimuli and monitoring the responses is done in a controlled sequence by test programs residing in the launch computers. These test programs must be thoroughly checked out before they are allowed to run at the launch facility. The rigid testing of the launch computer programs is done at simulation facilities that imitate, as closely as possible, the environment of the launch computers; i.e., the functions of the vehicle, the GSE, and the checkout system. Most of the checkout is performed with hardware simulators such as the Saturn V Breadboard, which uses partly actual flight hardware and simulates certain mechanical and hydraulic equipment by electrical circuits [1,2].

To aid the checkout engineer in the design and evaluation of test programs, two major software simulators have been developed by MSFC. These software simulators simulate the on-off functions of discrete networks by evaluating large sets of Boolean equations including discrete time-delays for pickup and dropout of relays, valves, etc. They evaluate the equations in non-real-time and are driven by predetermined sequences of states of switches and stimuli as generated by test programs [3,4].

More than three years ago, a joint effort between the Astrionics Laboratory and the Computation Laboratory of MSFC began to define a simulation system in which a digital computer would simulate, in real-time, the vehicle and GSE functions in response to stimuli originating from the two launch computers. The objective of this project was to find a new way to perform major functions of the Saturn V Breadboard with a more flexible digital computer, so that RCA-110A launch computer programs could be checked out, test programs could be evaluated, and the effect of malfunctions could be investigated without having to use and possibly damage expensive hardware. The primary design objectives were to insure that:

1. The simulator would act in such a way that the test programs of the two launch computers would think they were working with the actual vehicle and GSE in real-time.

2. The prime portion of the simulator, the software, should be structured in such a way that no reprogramming is necessary when a configuration other than Saturn V had to be simulated, as long as the hardware components could be described by the same nomenclature for the data base.

3. To provide the engineer with the capability to communicate directly with the computer when using this simulator.

It was determined that the feasibility of this approach could best be demonstrated by using the Saturn V configuration as a test bed.

The emphasis of this paper is on describing the software of the simulator. The operation of the simulator facility and the form of the mathematical models that are input into the computer are described in detail in Reference 5. However, to understand the structure and problem areas of the software, it is necessary to understand the configuration of the hardware also.

SCOPE OF SIMULATION

Hardware Configuration

A simplified diagram of the Saturn V Launch Computer Complex configuration is shown in Figure 1. The launch computers in the Launch Control Center (LCC) and in the Mobile Launch Facility (MLF) send out discrete signals (up to 2040 "Discrete Out or DO") to the vehicle through the GSE. The sequence and addresses of these signals are determined by the test programs. The vehicle then sends discrete and analog responses (measurements) back to the computers. Most of the discrete measurements (up to 3024 "Discrete In or DI," i.e., open/closed relay contacts, valves, switches, or gates) are fed through the GSE, while all the analog measurements and a few digital measurements are transmitted through the digital data acquisition system (DDAS) or telemetry system into the DDAS Computer Interface. The DDAS is the whole collection of equipment that lies between the sensors and the DDAS Computer Interface; i.e., a transmitter, a line driver and receiver, and a digital receiver station for each vehicle stage. The transmitter consists of a scanner, a digital and analog multiplexer, A/D converters, a generator of identification codes, and modulators; the line driver and receiver contain amplifiers and demodulators. The digital receiver station converts the demultiplexed measurement information into synchronized data words and address words and sends them to the Computer

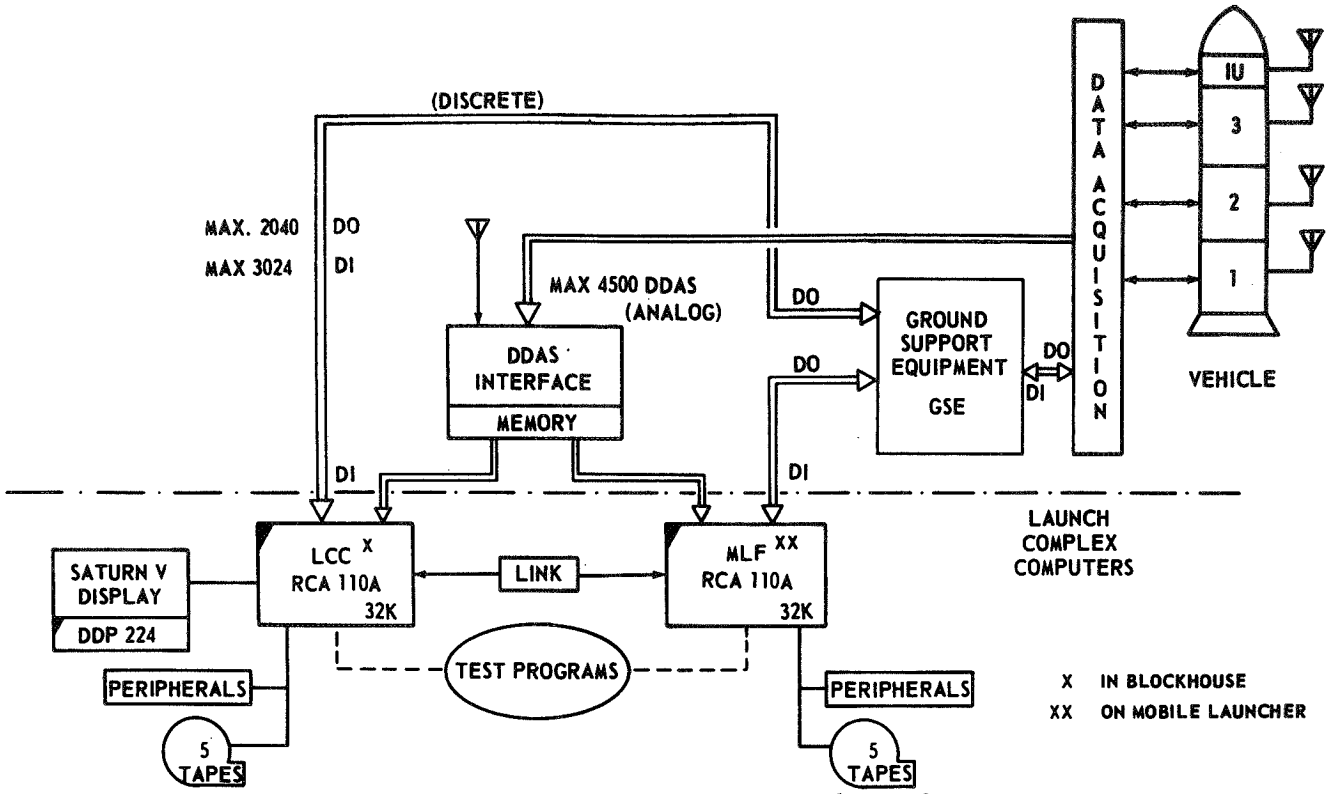


Figure 1. Saturn V Launch Computer Complex configuration.

Interface. The Computer Interface is mainly a digital memory that can store up to 8192 words and a special controller that stores one measurement word every 278 μsec in proper sequence and allows the launch computers to retrieve stored data at random under several modes. The data are stored in the Computer Interface according to their identification number containing the stage, channel, frame, multiplexer, and master frame numbers. The controller makes sure that the data request from the RCA-110A computer is properly decoded to find the requested measurement. For a measurement that has to be scanned at a higher rate, the scanner moving with constant speed accesses the sensor several times and, therefore, stores it at several places. The RCA-110A computer can access the data in the Computer Interface in several modes; e.g., the request can be synchronized with the incoming data or locked at a specific measurement. Up to 4500 DDAS measurements can be handled by the Computer Interface [6].

The launch computers themselves are connected through a data link for exchange of information. The test conductor controls the launch checkout through the Saturn V display, which is driven by a small DDP 224 computer. The coordination of the many test programs, display programs, and control programs for the peripheral equipment (printer, card reader, etc.) is done by the RCA-110A Operating System.

The simulator should perform the functions of the equipment shown in the upper portion of Figure 1. The hardware portion of the simulator comprises the SDS-930 digital computer with 32 000 words of core memory and its peripheral equipment and bulk memory devices and a special purpose interface (SDS Interface) that is similar to a small computer in size (Fig. 2). This interface performs the functions of the DDAS Computer Interface but does not contain a memory, since a portion of the SDS-930 memory is dedicated to these functions. The SDS Interface contains counters, special registers, and controllers that enable the two launch computers to communicate with the SDS-930 computer in the same modes as in their actual launch complex environment.

Data Base

The functions of the vehicle and its ground support equipment as seen by the test programs can

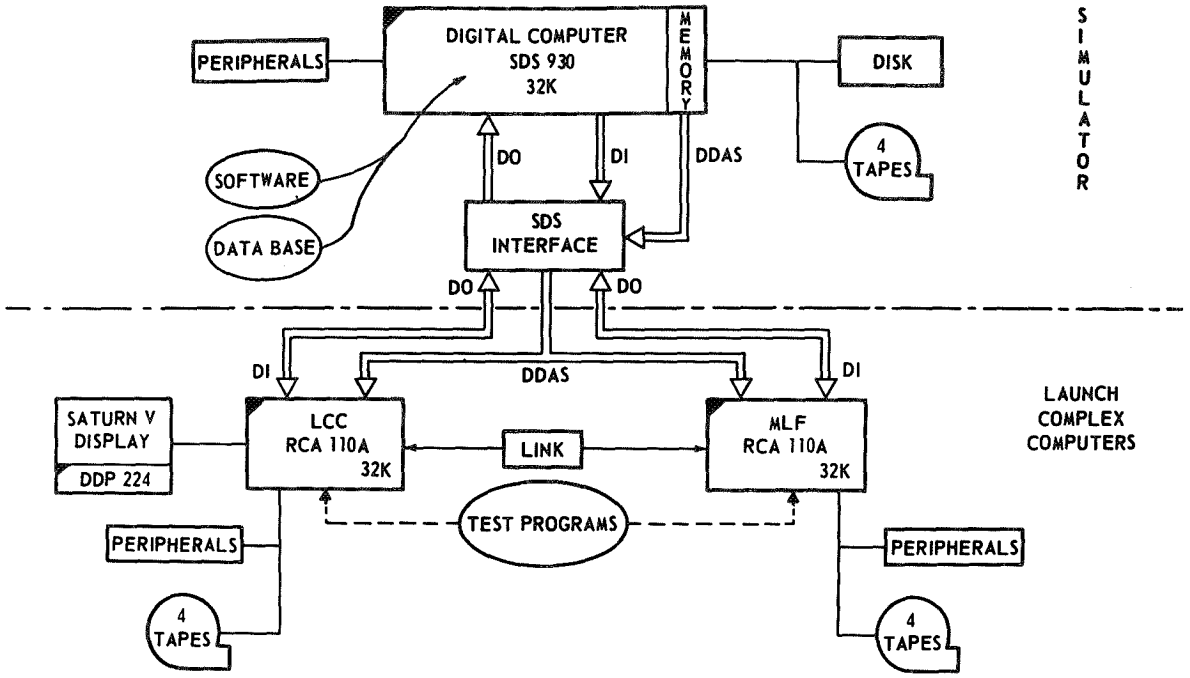


Figure 2. Real-time simulator computer systems configuration.

be described by large sets of logical equations and by analog time functions that are described by polynomials or tables. The logical or Boolean equations are time-dependent in the sense that they consider pick-up and drop-out time as a time-delay. The logical equations consist of AND and OR terms (* and +) and negations of a single variable (-). Special relays such as lock-out, lock-up, and latching relays can be expressed by equivalent circuits of regular relays. Figure 3 shows a simplified example of a typical discrete/analog circuit. For more detailed information, see References 5 and 7.

There are basically two types of equations possible:

1. Logical equation

$$E^{(i)} = Y_{11}^{(i)} * Y_{21}^{(i)} *\ldots* Y_{K_1 1}^{(i)} + Y_{12}^{(i)} * Y_{22}^{(i)} *\ldots* Y_{K_2 2}^{(i)} +\ldots+ Y_{1a}^{(i)} * Y_{2a}^{(i)} *\ldots* Y_{K_a a}^{(i)} ,$$

where

$$Y_{pq}^{(i)} = (-)\, Z_{pq}^{(i)}\left(P_{pq}^{(i)} , D_{pq}^{(i)} \right)$$

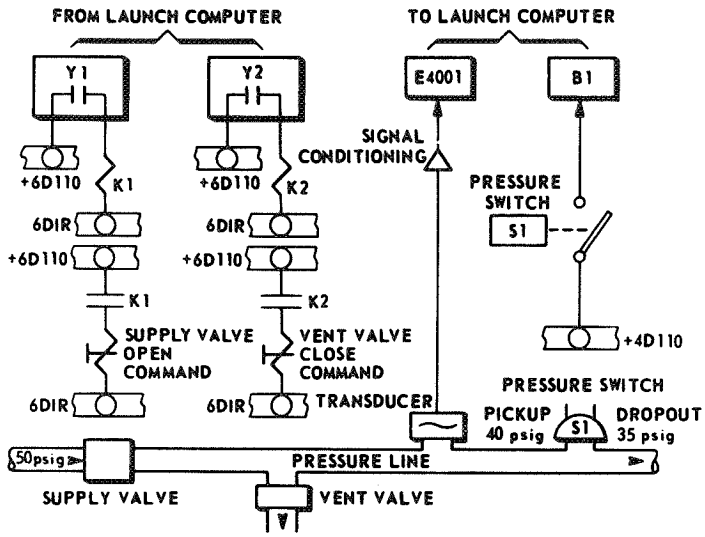
or

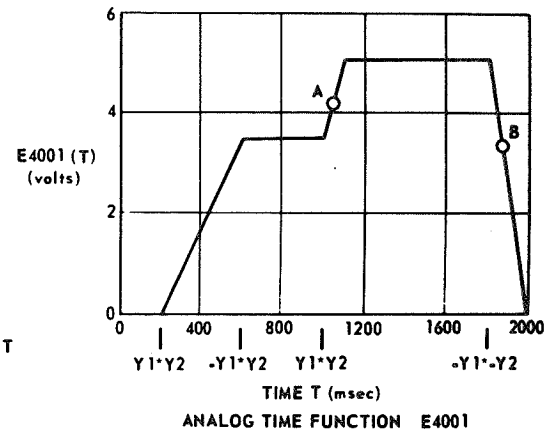
$$Y_{pq}^{(i)} = (-)\, Z_{pq}^{(i)} \,//\, P_{pq}^{(i)} , D_{pq}^{(i)} \,// \quad ,$$

$P_{pq}^{(i)}$ = pick-up time (amplitude) of element $Z_{pq}^{(i)}$,

$D_{pq}^{(i)}$ = drop-out time (amplitude) of element $Z_{pq}^{(i)}$,

and i, p, q, and a are unlimited index integers 1, 2, 3, ...; i.e., the number of equations, OR-terms, and AND-terms is not limited. Pick-up time for a relay is the time between activation of the coil and the closure of an associated contact. Drop-out time is the time between deactivation of the coil and the opening of an associated contact.





COMBINED LOGICAL/ANALOG EQUATION FOR SIGNALS TO COMPUTER:

ANALOG SIGNAL E4001 = 06110*Y1*Y2 /P, 10, 150, M5000, NO, M(500)/
+06110*-Y1*-Y2 /P, -25, 150, M50000, NO, M(200)/

LOGIC ANALOG

DISCRETE SIGNAL B1 = 04110*E4001//4000, 3500//

Figure 3. Example of a typical discrete/analog circuit.

Generally, pick-up time is the time-delay between cause and effect, and drop-out time the time-delay of the reverse action. Mathematically, if t_1 is the time instant of activation (deactivation), the discrete variable $Z_{pq}^{(i)}$ is

$$Z_{pq}^{(i)} = 0 \text{ for time } t < t_1 + P_{pq}^{(i)} \left(t \geq t_1 + D_{pq}^{(i)} \right)$$

$$Z_{pq}^{(i)} = 1 \text{ for time } t \geq t_1 + P_{pq}^{(i)} \left(t > t_1 + D_{pq}^{(i)} \right) .$$

For a relay that has a pick-up/drop-out time of less than 10 msec, the time-delay is ignored because the delay does not have an effect on slower mechanical devices. Thus, relay races between fast relays cannot be simulated, and it is not intended to detect them because the test programs do not check for them. For most relays, the bracket term can be deleted.

The value of a logical variable may depend on the amplitude of an analog value; e.g., the pressure in a line, instead of a time delay. Then, the terminators, //, are written instead of brackets.

The discrete variables can have different physical meaning. We distinguish between a "DO" (Discrete Output signal from RCA computer), "DI" (Discrete Input signal), digital DDAS, manual switch, power bus, and an "IV." An "IV" is an internal variable that is needed for internal computation when a circuit stores a signal.

2. Combined logical/analog equation

A logical equation can be combined with an analog function. Each analog function can be associated with only one OR-term of the logical equation. An analog function can be described in eight different ways as a polynomial, table, cyclic function, etc. [5]. In any case, the analog function is described to the computer by a one-letter code F

designating the type of format and a variant set of parameters, all of which are enclosed within the terminators, /.../. These parameters also contain the sampling rate for that particular analog variable, the maximum and minimum amplitude limits, the maximum time, and the period of the time function. The general form of a combined logical, analog equation is

$A^{(i)}$ = logical OR-term 1/F, analog function parameters 1/

+ logical OR-term 2/F, analog function parameters 2/

+ . . . . ,

where $A^{(i)}$ is an analog value.

The interpretation of this equation is as follows: Assuming that only one OR-term generates a "1" at one time (exclusively OR-terms), then the analog function of that particular OR-term is evaluated at the specified sampling rate.

The data base for the total Saturn V including the GSE amounts to about 15 000 equations of various length (Table 1). If other than Saturn V systems are to be simulated, another data base has to be established, but no reprogramming is necessary as long as the functions of the physical system can be described by the same types of equations. The data base is initially set up via the card reader; modifications of it and control commands for the simulation are input via teletypewriter.

TABLE 1. MAGNITUDE OF EQUATIONS FOR INSTRUMENT UNIT

		DO	SWITCH	DI	DDAS DI	BUS	IV	TOTAL
INSTRUMENT UNIT	Discrete Variables	247	557	791	186	69	579	2429
	Analog Variables							505
	Discrete & Analog							2934
	Discrete Equations			791	186	69	579	1625
	Analog Equations							505
ALL STAGES	Variables							about 10 000
	Max. Computer 4032 Capability		1000	6048		3000	3000	about 17 000
	Capability for DDAS (Analog)							4000

LENGTH OF EQUATIONS (Rough Estimate)

Smaller than 10	OR-terms	about 40%
Between 10 & 100	OR-terms	about 35%
Larger than 100	OR-terms	about 20%
Up to 1000	OR-terms	some %

SIMULATOR SOFTWARE

General

The software for the simulator can be divided into three major areas: (1) Interface Support Software, (2) Simulation Processor, and (3) Simulator Diagnostics. The Interface Support Software controls the input into the DDAS-tables of the SDS-930 core memory and the output from it to the RCA-110A computer supported by the hardware of the SDS Interface. It also controls the transmission of the DO's, DI's, and analog values; various counters; and clocks. The Simulation Processor generates the data base in the computer from the card input, evaluates the equations during the simulation run, and controls the selective printout of the simulation results (Fig. 4). The Simulator Diagnostics checks all hardware units of the SDS Interface such as counters, data link control signals, and data transfer registers, and the communication between the SDS-930 and RCA-110A computers. The diagnostics check especially for critical timing.

The design of the software is modular so that modifications can be made relatively easily. The total software excluding the simulator diagnostics consists of about 315 subroutines, which amounts to about 26 000 instructions. These figures should give a feel for the magnitude of the software effort.

Interface Support Software

As stated previously, the simulator must provide all data input values to the launch complex computers as those are provided by the launch vehicle, the ground support equipment, manual checkout panels, mechanical and pneumatic systems, etc. at the Saturn V Launch Computer Complex. Additionally, the simulator must accept input data from the launch complex computers and provide the necessary stimuli to the launch complex computers. These stimuli, in the form of DO signals, DI signals, and DDAS signals are provided by the interplay between the Interface Support Software and the SDS Interface.

Each discrete signal (DI and DO) is represented in fixed locations of the SDS-930 memory by the presence or absence of a single bit in the DI and DO Status Table, thus allowing 24 discretes to be represented in each 24-bit computer word.

Software is also provided to support the input/output requirements to direct access communication channels, time-multiplexed channels, etc. for the storage and retrieval of data from mass storage and the recording of data on magnetic tape.

DISCRETE OUT/DISCRETE IN SIGNALS

The Real-Time Phase of the Simulation Processor receives DO signals from either of the two RCA-110A computers via the SDS Interface and stores these signals in the DO Status Table. Upon receipt of the DO's, a chain of Boolean and DDAS equations is evaluted by the Real-Time Simulation Program, and the results of the evaluation are placed in either the DI Status Table or in the DDAS Data Table. Both the DI Status Table and the DDAS Data Table are scanned continuously by the SDS Interface, thereby providing current information to the RCA-110A computers upon request.

DIGITAL DATA ACQUISITION SYSTEM

The basic function of the DDAS facility at the Saturn V Launch Computer Complex is to periodically sample vehicle parameters and make this real-time data available to the two RCA-110A launch computers. In a simulated environment, the simulator and its associated interface hardware must commutate data for use by both RCA-110A checkout computers. This commutated data must be in the same format as the data provided by the Launch Computer Complex. This will allow the simulator to provide information for the RCA-110A through the commutation processing of the SDS Interface. The DDAS data represent both analog and discrete data. The analog data are represented in 10 bits of a 24-bit SDS-930 computer word. Each discrete is represented by the presence or absence of a single bit. There are ten discretes represented in each SDS-930 computer word.

DDAS simulation requires three DDAS memory tables within the SDS-930 computer for use by the simulator and the SDS Interface (Fig. 5). The DDAS words that are the results of the evaluation of the combined discrete/analog equations are stored in a block of memory of the SDS-930 computer that is referred to as the DDAS Commutation Data Table. The address of this data word is stored in the DDAS Commutation Address Table according to the sampling rate required for this measurement. As a final step in the commutation process, the data words are

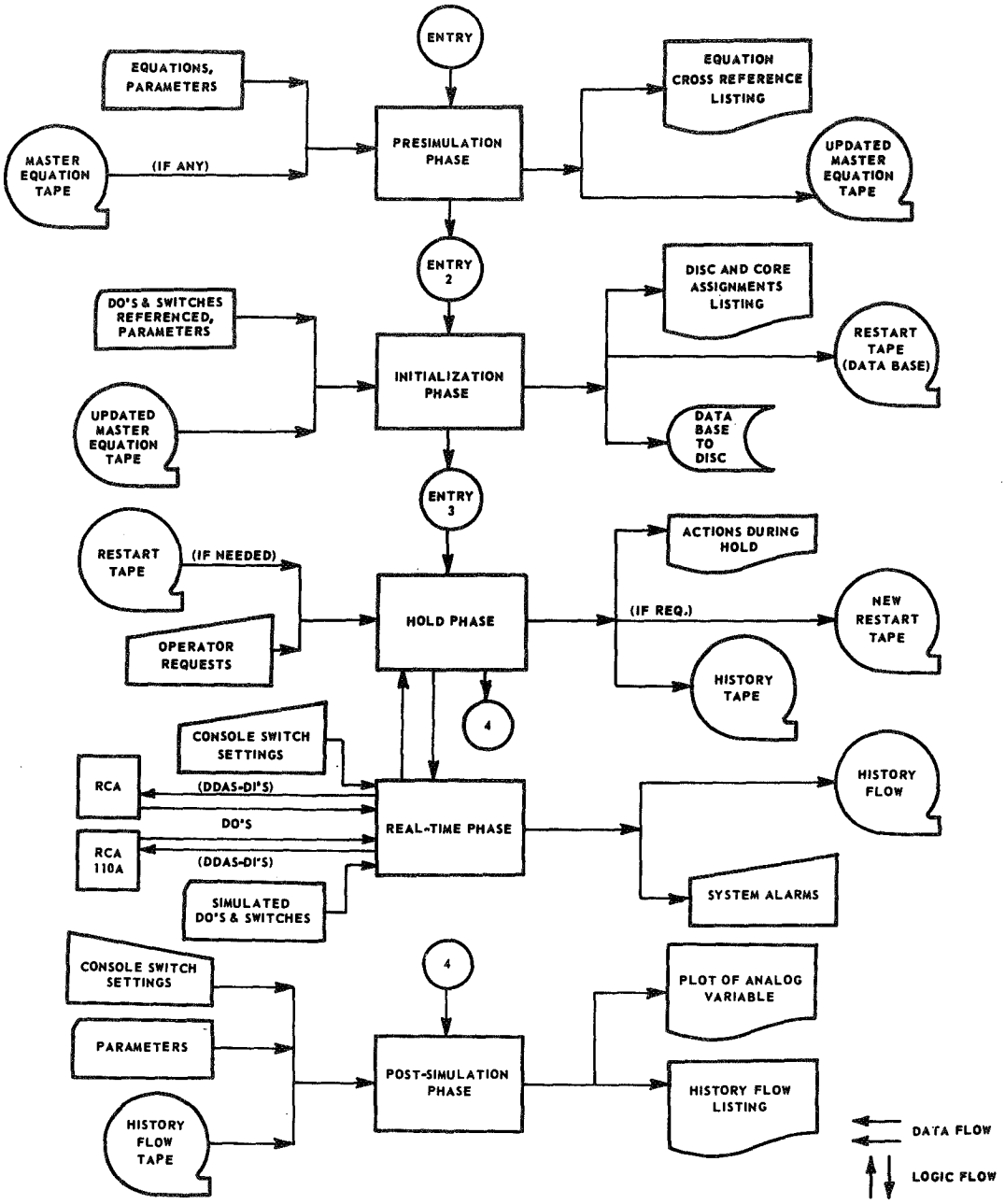
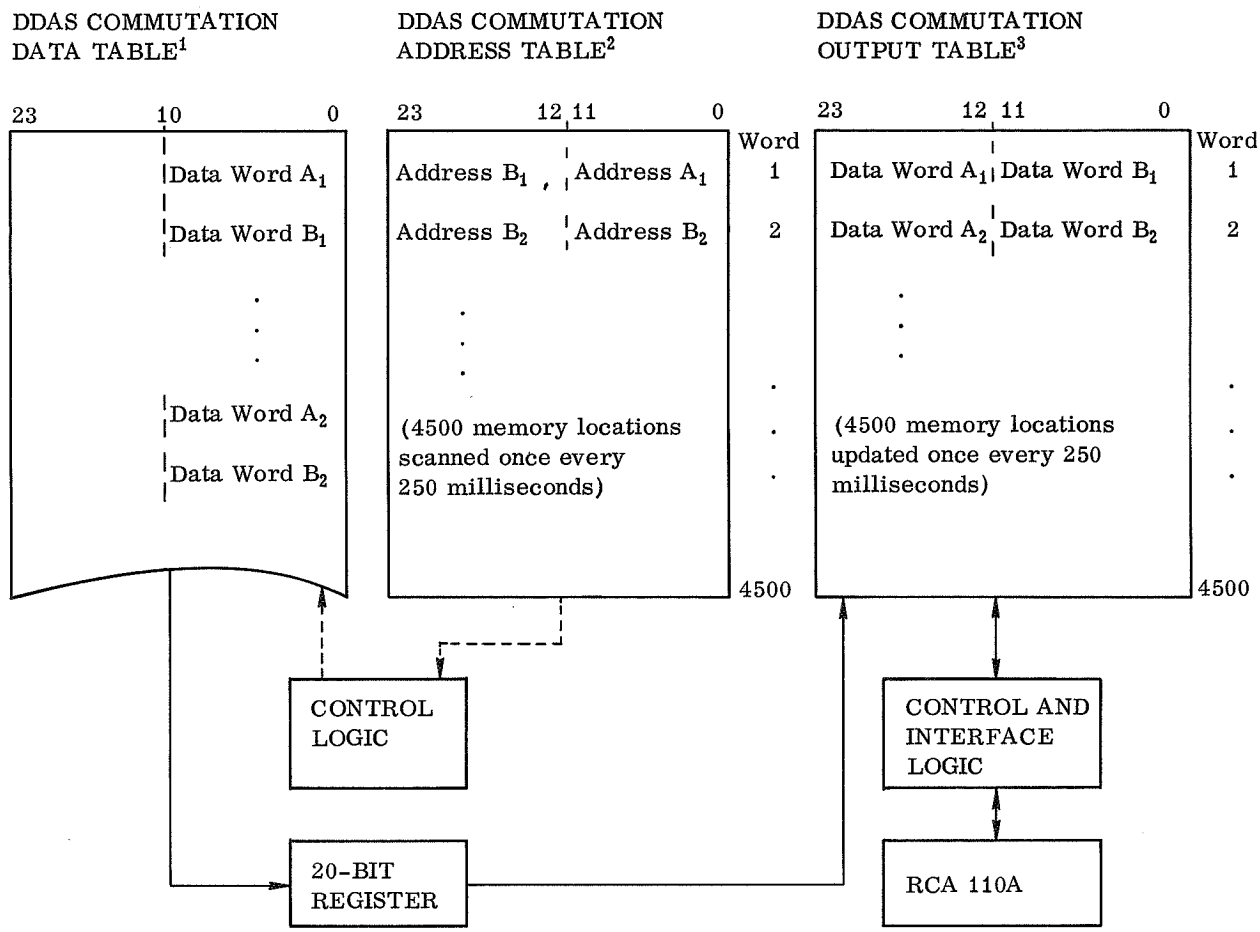


Figure 4. General flow of Simulation Processor.



1. Table is updated by the Simulator Program. Reserved during Initialization Phase.

2. Table is used by interface hardware for control of data transfer from Data Table to Output Table. Setup during Initialization Phase.

3. 24-bit words available to the RCA 110A's from this table. All accesses by the RCA 110A are through the interface hardware. Reserved during Initialization Phase.

Figure 5. DDAS Simulator commutation memory tables.

stored by the SDS Interface in the appropriate locations in the DDAS Commutation Output Table, where they can be accessed by either RCA-110A computer via the SDS Interface.

TIME COUNTERS AND CLOCKS

Two methods of timing are provided in the Simulation Processor to establish the necessary control for scheduling timed events during the Real-Time Phase of simulation.

A set of 190 elapsed time counters provides the capability for establishing arbitrary time-delays for scheduling of time-critical equation evaluations or re-evaluations. The counters are fixed, consecutive cells in the SDS-930 memory that are incremented by the interface hardware at 1-msec intervals. The number of counters is limited to 190, because incrementation of each counter requires two cycles (3.5 μsec) of memory access time (maximum of 0.665 msec, leaving 0.335 msec for DDAS commutation and for servicing RCA-110A data requests).

The Real-Time Simulation Phase stores the complement of the desired elapsed time in a predefined memory location and initiates the hardware incrementation which is then automatic until a cell counts to zero. When a cell has counted to zero, the automatic incrementation halts, the address of the zero cell is recorded by the SDS Interface, and a program interrupt occurs. The Interrupt Service Routine then schedules the associated equation for evaluation by placing it in the highest priority queue. The Interrupt Service Routine also removes this clock from the queue of active clocks and reinitiates the automatic incrementation. The automatic incrementation must be restarted within 1 msec to insure that all clocks are updated accurately.

The system real-time clock provides the relative time of events during the Real-Time Phase of the simulation processes. The real-time clock is serviced at 1-msec intervals by a system-generated interrupt. At each interrupt, the Real-Time Program increments the real-time clock.

DISC/TAPE-CORE COMMUNICATIONS

The Direct Access Communication Channel (DACC) controls the transmission of equations from the rapid access disc (RAD) to the SDS-930 memory. Initiation of the transmission is controlled by the Real-Time Simulation Phase, which uses the buffer code/disc address of the equation in conjunction with the Buffer Description Tables (Table 2) to compute the number of words to be transmitted and then initiates the transmission that remains under DACC control until completed. Upon completion, a program interrupt is generated. The Interrupt Service Routine transfers the equation to a specified memory location for later evaluation and initiates the transmission of the next equation to be input.

During the Real-Time Phase, the time-multiplexed communication channel is used for generating a complete history of events to magnetic tape.

Simulation Processor

GENERAL

The prime objective of the simulation processor is to prepare and execute the simulation in such a way that the stimuli of the RCA-110A computers are all received and their response signals (often several responses to one stimuli) are generated with the same precise time lag as in reality [8, 9]. The huge size of the equation data base (15 000 equations), the limited core size of actually only 13 000 words (19 000 of the 32 000 are used by the DDAS decommutator), the relatively long average access time of the disc memory (17 msec), and the minimum response time of some circuits to be simulated (about 100 msec) constrain the design of the Simulation Processor considerably. Considering the above constraints and in order to minimize the computation time during real-time simulation activity, as many functions as possible are performed in the Pre-Simulator Phase.

The Simulation Processor is divided into five major phases:

- Pre-Simulation Phase
- Hold Phase
- Initialization Phase
- Real-Time Simulation Phase
- Post-Simulation Phase

An overview of the general flow of the simulation processor is given in Figure 4.

PRE-SIMULATION PHASE

The Pre-Simulation Phase was designed to perform initial processing for all functions that could be predefined and established prior to execution of the Real-Time Phase. This approach was taken to simplify the real-time processing functions and to significantly reduce execution time and core memory space.

The Pre-Simulation Phase consists of 85 subroutines and approximately 11 200 instructions. The Pre-Simulation Phase consists of four subphases that are overlayed during execution in SDS-930 memory. A summary of Pre-Simulation Phase capabilities is shown in Table 3.

Phase 0 contains the utility and I/O subroutines for the remaining three subphases. This phase acts as the Pre-Simulation Phase monitor and remains in memory during execution of phases 1, 2, and 3 to control overlay and input/output operations.

TABLE 2. BUFFER DESCRIPTION TABLES

BUFFER CODE ADDRESS TABLE****

WORD	DEFINITION
0	ADDRESS OF START OF BUFFER GROUP WITH CODE 0
1	ADDRESS OF START OF BUFFER GROUP WITH CODE 1
2	ADDRESS OF START OF BUFFER GROUP WITH CODE 2
3	.
4	.
.	.
.	.
.	.
.	.
N	. N = 31

BUFFER CODE SIZE TABLE****

WORD	DEFINITION
0	NUM. OF LOC. ALLOTED TO EACH EQU. REC. IN BUFFER GROUP 0
1	NUM. OF LOC. ALLOTED TO EACH EQU. REC. IN BUFFER GROUP 1
2	NUM. OF LOC. ALLOTED TO EACH EQU. REC. IN BUFFER GROUP 2
3	.
4	.
.	.
.	.
.	.
N	. N = 31

BUFFER CODE NUMBER TABLE****

WORD	DEFINITION
0	NUMBER OF BUFFERS ALLOTED IN BUFFER GROUP 0
1	NUMBER OF BUFFERS ALLOTED IN BUFFER GROUP 1
2	NUMBER OF BUFFERS ALLOTED IN BUFFER GROUP 2
3	.
4	.
.	.
.	.
.	.
N	. N = 31

BUFFER CODE AREA TABLE****

WORD	DEFINITION
0	TOTAL NUMBER OF LOCATIONS IN BUFFER GROUP 0
1	TOTAL NUMBER OF LOCATIONS IN BUFFER GROUP 1
2	TOTAL NUMBER OF LOCATIONS IN BUFFER GROUP 2
3	.
4	.
.	.
.	.
.	.
N	. N = 31

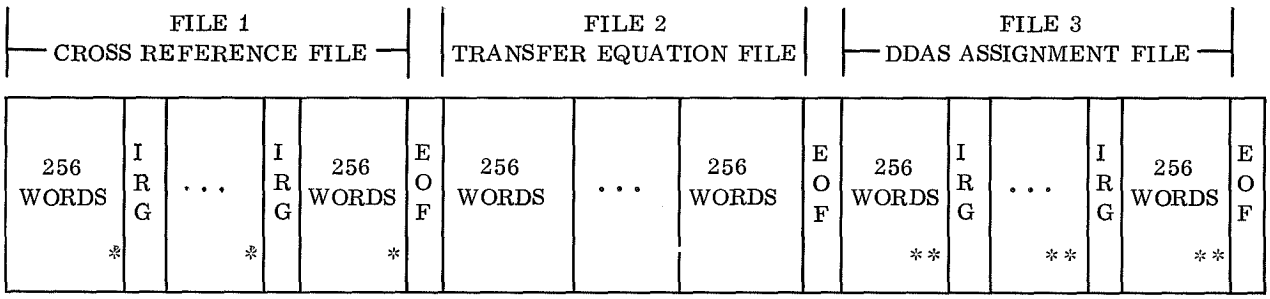
TABLE 3. PRE-SIMULATION PHASE CAPABILITIES

- Edits all equations input to the system.
- Builds a cross reference index of equations versus equation terms.
- Establishes tables for time-related equation terms.
- Arranges input equations into the proper equation calling sequence for real-time processing.
- Merges the cross reference values and the time-related values with the equation calling sequence.
- Using card input values, establishes an assignment file for all DDAS values.
- Produces a Master Equation Tape (magnetic tape) consisting of:
  a. Equation cross reference file
  b. Equation file
  c. Ordered DDAS assignment file
- Lists the Master Equation Tape.

The functions of the vehicle and the ground support equipment are described by Boolean equations that are punched on cards. These cards, containing the data base for the vehicle to be simulated, are input to phase 1 where the equations are edited. During this editing process, all equations that contain an error are listed on the line printer with each error flagged. The equations that are error-free are sorted and all duplicate equations are eliminated. The sorted, error-free equations are then used to update the Master Equation Tape (Fig. 6) which contains all equations that describe the vehicle configuration to be simulated. During the Master Equation Tape update, sorted equation cross-reference information is generated and output to magnetic tape for use in later processing.

Phase 2 of the Pre-Simulation Phase completes the development of the Cross-Reference File (Fig. 7) and the Transfer Equation File. This is accomplished by merging the cross-reference information with all related equations and generating the Cross-Reference File and the Transfer Equation File of the updated Master Equation Tape (Fig. 6).

All DDAS assignments are made by card input. The function of Phase 3 is to make the DRS, multiplexer, frame, and channel assignments from information contained on the input cards and develop the DDAS Assignment File (Fig. 8) of the updated Master Equation Tape. This information is ordered



The three files are composed of both physical and logical records. Physical records have an arbitrarily set word count of 256 words; logical records have a variable word count contained in the first word of the logical record. Each physical record may contain one or more logical records or only a 256 word portion of a logical record (as in File 2). Thus, a logical record may span one or more physical records (as in File 2).

* Reference Figure 7
** Reference Figure 8

EOF = End of File
IRG = Inter-Record Gap

Figure 6. Master Equation Tape format.

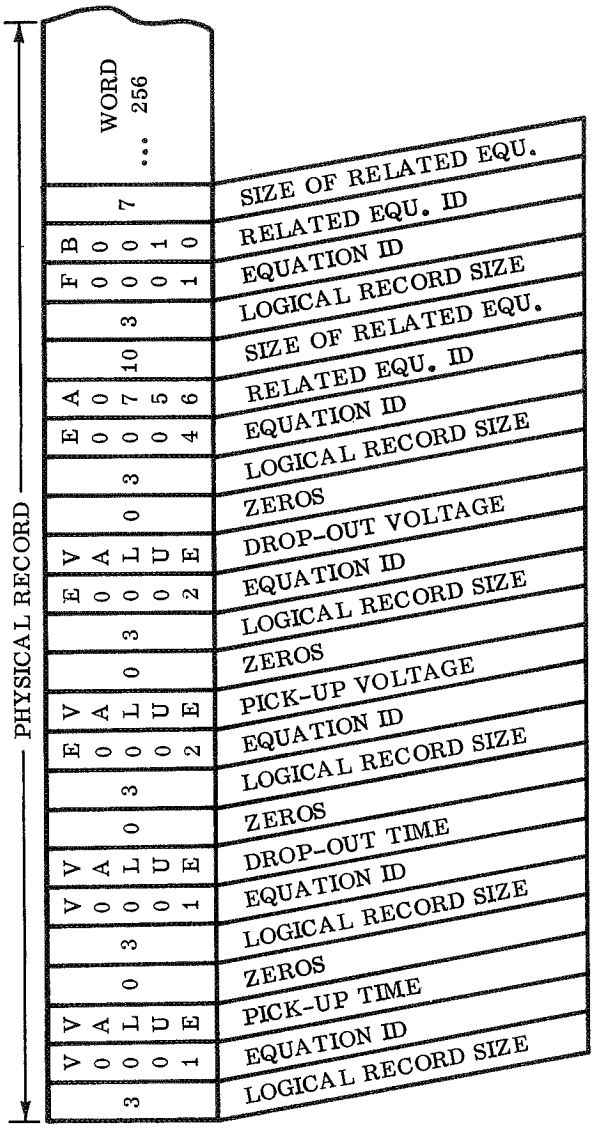
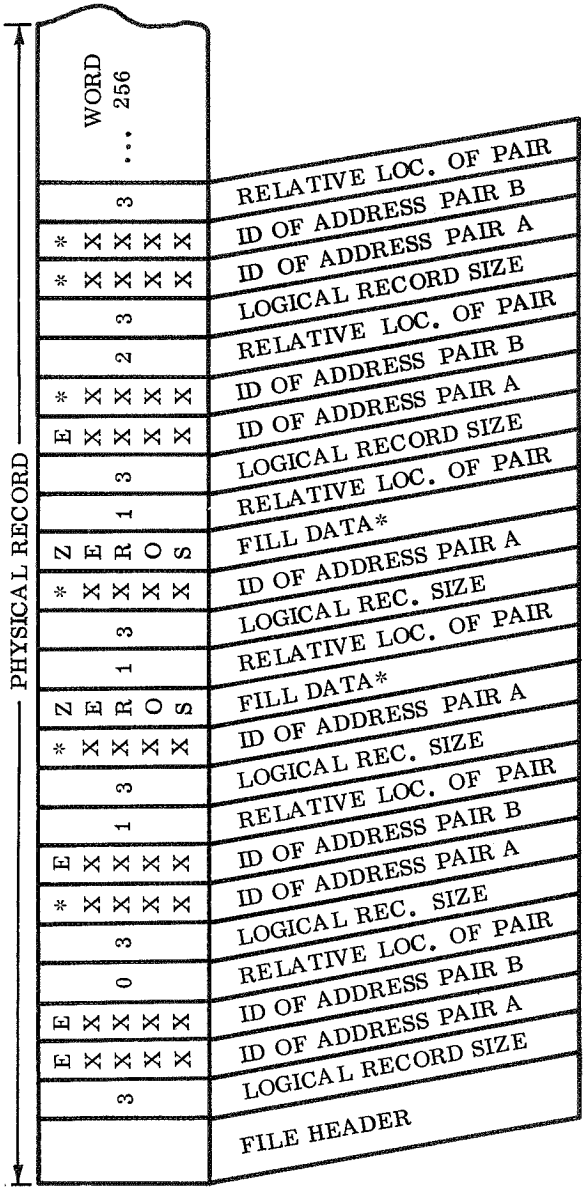


Figure 7. Cross-Reference File.

to conform to the requirements of the Initialization Phase, which develops the 4500-word commutation address table (Fig. 5).

INITIALIZATION PHASE

The Initialization Phase performs the final processing before real-time operations. It creates the proper environment in the SDS-930 computer and on the disc storage to initiate the simulation run. The simulated switches and DO's to be input during the Real-Time Phase are also input on punched cards during the Initialization Phase. The Initialization Phase utilizes these simulated variables and the Cross-Reference File of the Master Equation Tape to determine which equations will be active during



NOTE:

FOR SIDE A AND SIDE B ADDRESSES, ONLY ONE REAL DDAS ANALOG VARIABLE (E-CODE) MAY BE ASSIGNED TO THE SAME RELATIVE LOCATION WORD. A MAXIMUM OF TEN DISCRETE DDAS ANALOG (D-CODES) VARIABLES CAN BE ASSIGNED TO THE SAME RELATIVE LOCATION WORD.

*XXXX – BITS 0–9 CONTAIN THE DDAS BITS POSITION OF 1–10 FOR THE D-CODE VARIABLE. BITS 10–23 CONTAIN THE NUMERIC SUB-GROUP IDENTIFIER OF THE D-CODE.

E-CODE VALUE = 10 BIT DDAS DATA WORD

D-CODE VALUE = STATUS OF 1 BIT IN A 10-BIT DDAS DATA WORD

Figure 8. DDAS Assignment File.

the simulation run. Only the active equations will be reformatted and stored on the disc (Table 4). Initialization also produces a line printer listing of the active equations and all related cross-reference information (Table 5). To relate the simulated switches and DO's to the equations on disc during the simulation run, the Initialization Phase creates blocks of cross-reference information for the switches and DO's and transfers them to the disc (Table 6). Address tables, reference tables, data tables, status tables, history buffers for recording real-time events, and a 4500-word commutation table are dynamically allocated in-core and initialized. The remaining available core can then be assigned as equation buffer areas. To facilitate this assignment, a list of equation sizes versus the number of equations of that size is output to the teletype along with the location and size of core blocks available for use as equation buffers. Utilizing the above information, the user determines the optimum number of buffers and buffer sizes and provides the information on cards for input by the Initialization Phase. All equation buffer information required by the Real-Time Phase is arranged by the Initialization Phase into the four buffer description tables (Table 7).

HOLD PHASE

The Hold Phase program provides a convenient transition to real-time activities either initially or following a hold or suspension of a prior real-time

TABLE 4. FORMAT OF EQUATION ON DISC

WORD	DEFINITION
0	NUMBER OF WORDS IN EQUATION RECORD (P+1)
1	DISC ADDRESS ASSIGNED THIS EQUATION RECORD
2	EQUATION MNEMONIC ID
3	REF. TO PICK-UP TIMES (REL. ADD. J FROM WORD 2)
4	REF. TO DROP-OUT TIMES. (REL. ADD K FROM WORD 2)
5	REF. TO PICK-UP VOLTAGES (REL. ADD. L FROM WORD 2)
6	REF. TO DROP-OUT VOLTAGES (REL. ADD. M FROM WORD 2)
7	REF. TO RELATED EQUA ID'S (DISC. ADDRESSES) (REL ADD. N)
8	FIRST WORD OF ACTUAL EQUATION
9	2ND WORD OF ACTUAL EQUATION
.	.
.	.
.	.
.	LAST WORD OF ACTUAL EQUATION ($ OPERATOR)
J	1ST PICK UP TIME
J+1	2ND PICK UP TIME
.	ETC.
K	1ST DROP-OUT TIME
K+1	2ND DROP-OUT TIME
.	ETC.
L	1ST PICK-UP VOLTAGE
L+1	2ND PICK-UP VOLTAGE
.	ETC.
M	1ST DROP-OUT VOLTAGE
M+1	2ND DROP-OUT VOLTAGE
.	ETC.
.	
.	
.	
N	BUFFER CODE (BITS 0-4) RELATED EQ. DISC ADD. (BITS 5-23)
N+1	BUFFER CODE (BITS 0-4) RELATED EQ. DISC ADD. (BITS 5-23)
.	BUFFER CODE (BITS 0-4) RELATED EQ. DISC ADD. (BITS 5-23)
P	BUFFER CODE (BITS 0-4) RELATED EQ. DISC ADD. (BITS 5-23)

TABLE 5. EQUATION CROSS-REFERENCE LISTING

X0000		E0001	E0002					
X0001		E0002						
X0002		E0003	F0026					
X0007		A0004						
X0008		E0004						
X0009		E0004						
X0010		F0016						
X0011		F0026						
X0012		F0026						
X0013		E0016						
X0014		F0016						
X0015		E0016						
X0016		F0017						
X0017		F0027						
X0018		E0018	E0028	F0018	F0028			
X0019		E0019						
X0020		E0019						
X0021		F0111						
X0022		F0211						
X0023		V0112	F0012					
X0024		V0112	E0029	F0012				
S0001		01110	06020	06100				
S0002		A0001						
S0009		V0011	V0012	V0112	V1000	V2222	V2500	A0001	A0004
		06020	06100	D1600	D2500	D3000	D4200	D5050	E0001
		E0007	E0009	E0010	E0011	E0016	E0018	E0019	E0028
		F0026	F0027	F0028	F0111	F0211			
V0011		E0011							
V0012		A0011							
V0112		V0012							
V1000		E0006							
V2222		E0007							
V2500									
A0001		E0003							
A0004		E0005							
A0005									
A0006									
A0007									
A0011									
01110		V0012	A0005	A0007	A0011	D3000	E0007	E0011	E0018
06020		A0006	E0003	E0004					
06100		A0001	A0004	D1600	D4200	D5050	E0002	E0006	
D1600									
D2500									
D3000									
D4200									
D5050									
E0001	DV	2999.00							
	PV	4098.00							
F0111		V0011							
F0211		V0011							

TABLE 6. SWITCH AND CROSS REFERENCE BLOCK

FORMAT OF SWITCH CROSS REFERENCE BLOCK

WORD	DEFINITION
0	NO. OF ENTRIES (6+K+3*M)
1	BUF. CODE/DISC ADDR OF THIS BLOCK
2	NO. OF ENTRIES/REL. ADDR OF ORIG. IDS
3	NO. OF ENTRIES/REL. ADDR. OF REASSIGNED IDS
4	NO. OF ENTRIES/REL. ADDR. OF RANGES
5	/REL. ADDR. OF RELATED IDS
6	BUF. CODE/DISC ADDR OF 1ST RELATED ID
7	BUF. CODE/DISC ADDR OF 2ND RELATED ID
.	ETC.
5+K	BUF. CODE/DISC ADDR OF 1ST RELATED ID
6+K	1ST ORIG. ID
7+K	2ND ORIG. ID
.	ETC.
5+K+M	M(TH) ORIG. ID
6+K+M	START REL ADDR/END REL ADDR OF IDS RELATED TO 1ST ORIG
6+K+M	START REL ADDR/END REL ADDR OF IDS RELATED TO 2ND ORIG
.	ETC.
5+K+2M	START REL ADD/END REL ADD OF IDS RELATED TO M(TH) ORIG
6+K+2M	1ST REASSIGNED ID
7+K+2M	2ND REASSIGNED ID
.	ETC.
5+K+3M	M(TH) REASSIGNED ID

WHERE M=NO. OF ORIGINAL IDS
K=NO. OF RELATED IDS FOR ALL ORIGINAL IDS

FORMAT OF DO CROSS-REFERENCE INDEX BLOCK

WORD	DEFINITION
0	NO. OF ENTRIES (3+M+R1+R2...+RM)
*	BUF. CODE/DISC ADDRESS OF THIS BLOCK
2	REF. TO START OF RELATED IDS (3+M)
3	NUMERIC PORTION OF 1ST DO ID/REL. ADDR. OF LAST REL. ID
4	NUMERIC PORTION OF 2ND DO ID/REL. ADDR. OF LAST REL. ID
.	ETC.
2+M	NUMERIC PORTION OF M(TH) DO/REL ADDR OF LAST REL. ID
3+M	BUF. CODE/DISC ADDR. OF 1ST DO RELATED TO DO OF WORD 3
4+M	BUF. CODE/DISC ADDR OF 2ND DO RELATED TO DO OF WORD 3
.	ETC.
2+M+R1	BUF. CODE/DISC ADDR OF R1(TH) DO RELATED TO DO OF WORD
3+M+R1	BUF. CODE/DISC ADDR OF 1ST DO RELATED TO DO OF WORD 4
4+M+R1	BUF. CODE/DISC ADDR OF 2ND DO RELATED TO DO OF WORD 4
.	ETC.

TABLE 7. INITIALIZATION PHASE CAPABILITIES

- Extracts the specific equations to be active from the cross-reference file of the Master Equation Tape utilizing information input from cards.
- Prints a cross-reference listing of all variables to be active.
- Outputs a summary of number of equations versus equation length.
- Outputs the location and size of core blocks available for use as equation buffers.
- Assigns a disc address to each active equation.
- Reformats each active equation and its cross-reference data and stores the combined results on disc.
- Prints a listing of all equations stored on the disc.
- Creates cross-reference data for all active DO's and switches in a format usable in real-time. This is also stored on the disc.
- Assigns core locations for all necessary tables.
- Assigns core locations to be used as equation input buffers.
- Initializes all tables and buffers.

simulation and gives the user the capability to control the environment of a specific test. He utilizes this capability by issuing commands to the Hold Phase Program via the teletype, which initiates execution of predefined functions (Table 8). He may request the current value or status of any active variable, or perform debugging activities such as dumping any area of core or disc.

A very important option is the capability to create a restart tape. This tape contains an image of core memory and disc memory as it was established at the end of the Initialization Phase. Thus, the restart tape provides the means by which the same or a similar real-time simulation can be executed without repeating the Initialization Phase processes. The Real-Time Simulation Phase overlays the Hold Phase in memory when the user issues the "T" directive (Table 8).

A list of Hold Phase capabilities is given in Table 9.

REAL-TIME SIMULATION PHASE

The Real-Time Simulation Phase drives the SDS-930 computer and the SDS Interface (Fig. 2) for the actual simulation of the launch vehicle complex. Operation during the Real-Time Phase also requires the execution of the launch computer programs, operating asynchronously, in the RCA-110A computers. Collectively, the three computers, their operating software, the interface hardware, and the launch complex/vehicle mathematical model provide the principal ingredients for the simulation of a Saturn V launch vehicle complex (Fig. 2).

The simulation process consists primarily of: (1) equation evaluation triggered by inputs from the launch computers, by internally generated values, and from the card reader by manual input; and (2) outputs developed as a result of equation evaluations that are made available to the SDS Interface for use by the launch computers.

The communication between the launch computers (RCA-110A) and the Simulator (SDS-930) is through the SDS Interface. This communication is in the form of discrete signals sent from either RCA-110A to the SDS-930 (DO), discrete signals requested by either RCA-110A from the SDS-930 memory (DI), and DDAS analog and discrete signals available at the SDS Interface for the RCA-110A computers to access when desired. Simulated DO's and switch

TABLE 8. HOLD PHASE COMMANDS

Hold Phase directives (commands) and options requested are as follows:

DIRECTIVE CHARACTER	OPTION REQUESTED
I	Print Hold Phase Instructions
N	Write Memory on New Restart Tape
R	Restore Memory from Previous Tape
P	Proceed to Post-Simulation Phase
T	Proceed to Real-Time Simulation Phase
F	Print F-Type Data Table
E	Print E-Type Data Table
U	Update Data Value or Update Status
Q	Dump Disc on Line Printer
V	Write Memory and Disc on Restart Tape
D	Restore Memory and Disc from Restart Tape
S	Print Summary of True Status
B	Branch to 32K Debug
C	Exit Debug

settings can be input to the Real-Time Simulation Phase from the card reader by operator action.

Upon receipt of a Discrete Output from the SDS Interface, the Real-Time Simulation Phase will schedule a chain of Boolean equations for evaluation. Some equations must be evaluated when there is a change in status of a dependent variable, some other equations must be evaluated at regular time intervals or at the end of a set time period, and still others must be evaluated when a specific analog variable reaches a particular value.

TABLE 9. HOLD PHASE CAPABILITIES

- Record memory in an initialized state on a magnetic tape.
- Restore memory to an initialized state from a previously created tape.
- Record memory in an initialized state and the disc contents on a magnetic tape.
- Restore memory and the disc contents to an initialized state from a previously created tape.
- Print the status of any variable within the system.
- Change the status of any variable within the system.
- Advance the simulator to the real-time phase.
- Advance the simulator to the post processing phase.
- Utilize the capabilities of the SDS Program, 32K DEBUG.

Each equation that is to be evaluated must be retrieved from the disc where it was stored during the Initialization Phase. Each equation record contains information required for evaluation, such as pick-up and drop-out times, pick-up and drop-out values, and equation identifiers of related equations. The identifier or ID of an equation is the name used to refer to the dependent variable on the left side of an equation.

In general, each DO received from the SDS Interface will require an equation evaluation that, in turn, generates a response (DI) to be supplied to the SDS Interface that makes the response available to the RCA-110A computers. During equation evaluation, DDAS responses are also developed and provided to the SDS Interface for DDAS commutation to the RCA-110A computers. The generation of a response may require the evaluation of several dependent equations. An example should explain the evaluation process [10]. It is assumed that the following three logical equations are given:

$$Y_1 = Y_3 + DO*Y_1 + DO*Y_2$$

$$Y_2 = DO*Y_1*Y_3 + Y_2*\overline{DO} + Y_2*Y_1$$

$$Y_3 = \overline{Y}_1*\overline{DO}*Y_2 + \overline{Y}_1*DO*\overline{Y}_2 + \overline{Y}_1*Y_3 + Y_3*DO$$

The three variables Y_1, Y_2, and Y_3 appear on the left side of the equations and also on the right side of the equations. If the external variable DO changes state from "1" to "0," the first evaluation of the three equations yields the intermediate result $Y_1 = 1$, $Y_2 = 0$, $Y_3 = 1$. The next evaluation of the same equations using these intermediate results yields the new values $Y_1 = 1$, $Y_2 = 0$, $Y_3 = 0$. This process of successive evaluation is continued until a stable state is reached; i.e., no change of state of any variable. In this example, four evaluations have to be performed. If no stable state can be reached, the circuit oscillates. This case occurs in the example if another initial condition is used where the fourth evaluation yields again state I. Table 10(a) illustrates the evaluation process, and Table 10(b) depicts all possible states of the example.

As each equation evaluation is completed, the necessary information with which to schedule the evaluation of all affected or related equations must be immediately available. This is done by having all related equation ID's available as each equation is brought in from disc and by logging all related equation ID's into the appropriate queue (Table 11). It is obvious from this example that the processing of many equations within the Real-Time Phase can become time-critical.

Throughout the Real-Time Phase execution, a complete record of events is recorded on the History Tape. This magnetic tape contains information such as equation ID's; times a state changed; and in the case of analogs, the actual value of the variable at a given time. This tape is used as input to the Post-Simulation Phase to record on the printer the complete history of the particular simulation run.

A list of capabilities of the Real-Time Phase is shown in Table 12 and a simplified diagram of the overall processing is shown in Figure 9.

POST-SIMULATION PHASE

The Post-Simulation Phase processes the Real-Time Phase history tape according to options selected from the SDS-930 console. The test conductor is provided the following options that he can choose via the switches:

1. A listing of all events on the history tape

2. A listing of only events where a change in status or value occurred

3. A line printer plot of up to eight analog variables (analog value versus time)

The user also specifies (via card input) which system variables (1 to 10) he desires to be printed and the relative time of the first and last event to be printed (nominally 0 to 24 hours). This card input is then used in conjunction with options 1 or 2 above to provide the particular listing desired.

A complete list of Post-Simulation Capabilities is shown in Table 13.

Simulator Diagnostics

GENERAL

The functional design of the acceptance and diagnostic program not only provides a method for thorough checkout of the SDS-designed interface equipment, but also results in an effective and permanent system of diagnostic procedures [11, 12]. Modularity, simplicity, and thoroughness comprise the basic philosophy used in the design of the acceptance test procedures. To simplify system programming problems, the acceptance test and diagnostic programs function within the framework of the standard capabilities provided by the RCA-110A TAME System (assembler, library routines, utility routines, and loader).

The order in which the diagnostic tests are performed follow the order as outlined in the design specification [13, 14]. This is to insure that certain interface equipment components are working correctly before that component is used in another test. The functional design of the system, however,

TABLE 10(a). RESULTS OF SUCCESSIVE EVALUATION OF LOGICAL EQUATIONS WITH TWO DIFFERENT INITIAL CONDITIONS IF DO CHANGES STATE FROM "1" TO "0"

Logical equations:

$$Y_1 = Y_3 + DO^* Y_1 + DO^* Y_2$$

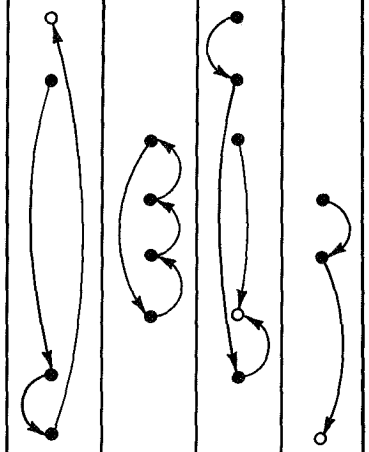
$$Y_2 = DO^* Y_1^* Y_3 + Y_2^* \overline{DO} + Y_2^* Y_1$$

$$Y_3 = \overline{Y}_1^* \overline{DO}^* Y_2 + \overline{Y}_1^* DO^* \overline{Y}_2 + \overline{Y}_1^* Y_3 + Y_3^* DO$$

Variable (= Equation) Name	Initial State	No. of Evaluation I	II	III	IV	Initial State	No. of Evaluation I	II	III	IV
Y_1	0	1	1	0	0	1	1	0	0	1
Y_2	0	0	0	0	0	1	1	1	1	1
Y_3	1	1	0	0	0	1	0	0	1	0
		Stable transition (1)					Unstable transition (2)			

TABLE 10(b). TRANSITION TABLE FOR ALL POSSIBLE STATES

Y_1	Y_2	Y_3	DO = 0	DO = 1
0	0	0		
0	0	1		
0	1	1		
0	1	0		
1	1	0		
1	1	1		
1	0	1		
1	0	0		
			(1) (2)	(3) (4)



● denotes transition state

○ denotes stable state

(1) (3) (4) stable transition

(2) unstable transition

TABLE 11. QUEUE FORMAT

WORD	DEFINITION
0	NUMBER OF CELLS IN QUEUE
1	POINTER TO NEXT AVAILABLE LOCATION (4 TO N)
2	POINTER TO EQUATION BEING EVALUATED (3 TO N)
3	POINTER TO LAST EQN. BROUGHT INTO CORE (3 TO N)
4	FIRST ENTRY BUFFER CODE AND DISC ADDRESS
5	SECOND ENTRY .
.	. .
.	. .
N	LAST ENTRY PRIOR TO CYCLING TABLE

TABLE 12. REAL-TIME PHASE CAPABILITIES

- Responds to DO inputs from the RCA-110A's.
- Responds to SWITCH inputs from the card reader.
- Responds to simulated DO inputs from the card reader.
- Responds to system interrupts — millisecond clock, elapsed timer and system error interrupts.
- Responds to disc interrupts.
- Evaluates appropriate Boolean equations.
- Maintains tables of all conditions of the system.
- Maintains clocks for various timing processes.
- Records all variable changes and evaluation results of the system on magnetic tape.
- Terminates real-time functions upon command when the system becomes stabilized.

provides complete test independence. Therefore, the order of tests may be selected at random to facilitate rapid location of a suspected malfunction.

TABLE 13. POST-SIMULATION PHASE CAPABILITIES

- Time span — time to begin and end printing
- Variables to be listed

Only the changes in value or status of variables on the history tape will be printed unless the test conductor specifies that all information be printed by setting a "BREAKPOINT" on the SDS-930 console.

A secondary function of the post processor is to provide a line printer plot of any analog variable within the system. The value of the analog variable is plotted against time. Again, the test conductor is allowed to specify the time span and the analog variable(s) to be plotted.

Since the diagnostic programs operate within the RCA-110A, the standard RCA-110A main frame diagnostics are used to insure that the RCA-110A main frame, I/O data trunks, and all peripheral equipment (excluding the SDS Interface) are operating properly.

SDS-930 — RCA-110A COMMUNICATIONS

To effectively control and coordinate the tests and diagnostic programs, the RCA-110A issues command words (DO's) that consist of a predefined octal bit configuration to the SDS-930 via the Discrete Output Channel. Once the command word is received by the SDS-930, it is interrogated to determine which test should be initialized. When the interrogation procedure is satisfied, a response word is sent back to the RCA-110A via the Discrete Input Channel.

TEST CONTROL LOOP

The Test Control Loop is the system monitor that controls the scheduling of tests. It monitors the various sense switches and transfers control to the requested diagnostic subprogram when the appropriate switch is toggled.

Each diagnostic subprogram has the ability to print a list of Predicted versus Received data (PRED/RCVD) for each error. This option is

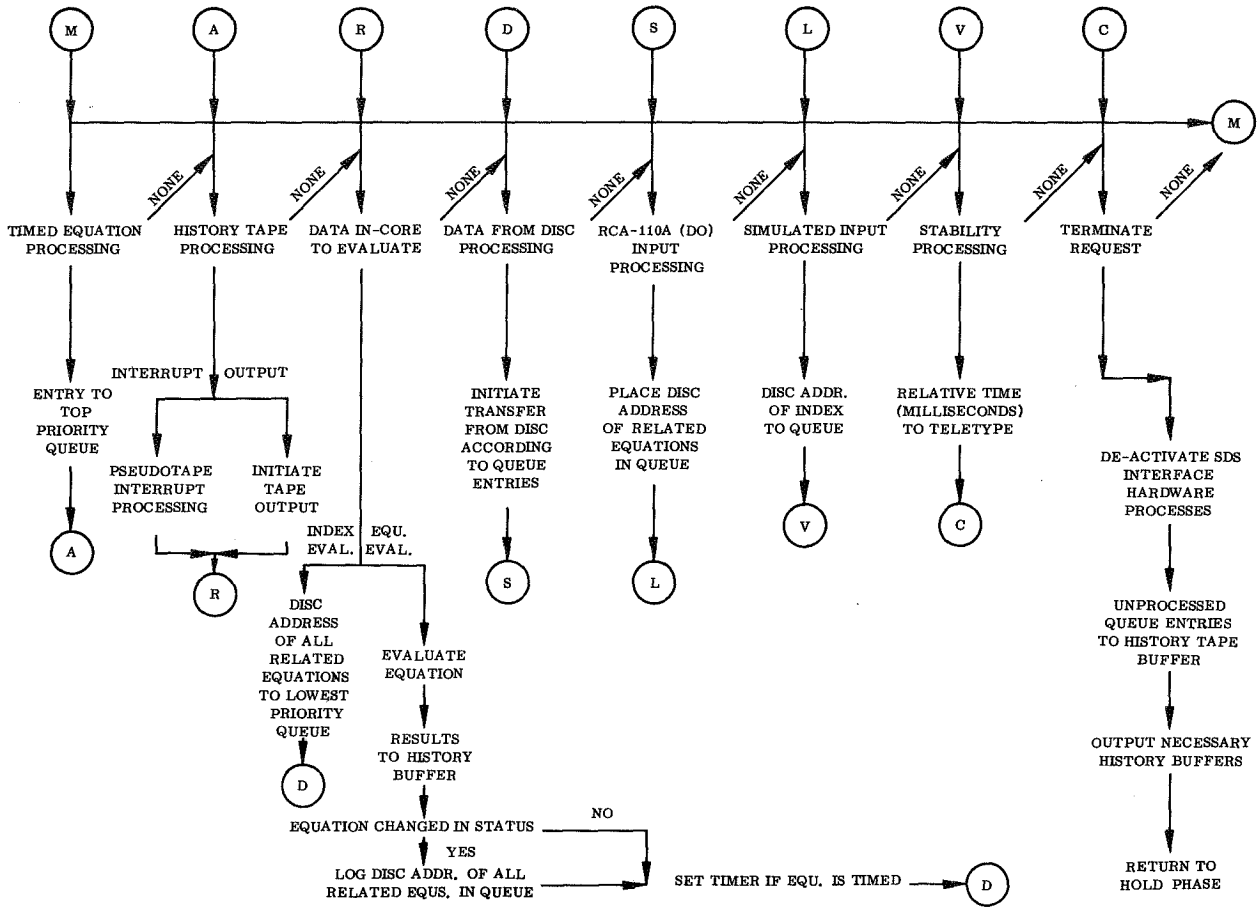


Figure 9. Simulator flow diagram.

selected during monitor control by toggling a sense switch on the RCA-110A control panel. It causes the diagnostic subprogram to list all interface data errors (predicted value and actual erroneous values received) encountered by the data validity checks performed within each diagnostic subprogram. Control may be transferred to the monitor at any time by setting a priority request switch.

An option for dumping raw data exactly as it was received from the SDS Interface is available to the user by setting a sense switch before each diagnostic test is executed. Another sense switch assignment permits automatic looping of a test.

The Discrete Output Diagnostic is designed to insure the communication link between the RCA-110A and SDS-930 computers via the DO data channel is operating properly. Since the DO data channel is used for systems test control, the validity of this link must be tested before any other tests are conducted. Five unique bit configurations are transmitted via the DO data channel to the SDS-930 and are immediately strobed back into the RCA-110A.

The Discrete Input Diagnostic checks the two DI data channels between the RCA-110A and SDS-930 computers. All modes of DI scan are executed by transmitting a unique bit configuration to the SDS-930 via the DO data channel followed by the DI scan that gathers the same data back into RCA-110A core memory (Status and Log Tables). Each bit configuration transmitted is compared to each bit configuration received. Data channels are also checked for parity error, inoperative, etc.

The Multiple Operand Address Diagnostic (MTOAD) assures that the addressing of more than one group of DO lines will cause an error condition and automatically inhibit further DO transmission.

The test consists of generating successive pairs of illegal addresses until all addresses have been made; e.g., 1-2, 2-3, etc. Between each MTOAD condition, an attempt to transmit a legal address is made as a check on the automatic inhibit caused by the MTOAD. When the legal address fails transmission, the interface is reset, and the next successive MTOAD is generated.

The time signals used with the discrete signals include those of the Eastern Standard Time and the Relative Time Counter. The Relative Timer Diagnostic checks the bit incrementation of the Relative Time Counter (27 bit, 1 msec). This counter supplies the means for having a zero time reference in the computer that can be reset under program control.

The EST Timer Diagnostic checks for proper setting and incrementation of the Gray Code counter. Preset Gray Code values are sent to the SDS Interface via the DO data channel. If the preset value is not returned or is returned in error, two more attempts are made before an error message is printed. Upon detection of the proper preset value, the RCA-110A checks for proper incrementation. Any errors detected are logged with the expected time and the received time.

The Elapsed Timers Diagnostic is initiated by the RCA-110A, but the main functions of the diagnostic checks are performed within the SDS-930. (See the topic entitled Time Counters and Clocks.) After the first timer has counted to zero, an interrupt occurs, whereupon the elapsed timers are interrogated and compared to timing criteria within the discrete and DDAS activities.

The first part of the Digital Data Acquisition System Diagnostic assures that commutation occurs at a 3.6-kc rate. This test is initiated by the RCA-110A but is performed by the SDS-930 program that cycles through and updates each DDAS data word once every 250 msec. After the commutation test is completed, the RCA-110A starts the DDAS Interface Test, which consists of exercising the DDAS Interface to insure proper commutation on all Digital Receiving Stations (DRS), and verifying valid data and time responses for all modes and submodes of DDAS scans.

The ACE Interface Diagnostic checks for proper transfer of data words between the RCA-110A and the SDS-930 computers via the ACE equipment by comparing four data words with different bit patterns.

TOTAL DYNAMIC SYSTEM TEST

This test is designed to exercise all interface equipment between both RCA-110A computers and the SDS-930 computer. The programs are executed in such a manner that peak data transmission operations occur between the three computers.

Either of the two RCA-110A computers may initiate the test by transmitting the appropriate command word. The SDS-930 computer initializes its discrete system, commutation is started on all five Digital Receiving Stations, and the Eastern Standard Clock is preset to zero. All three computers (both RCA-110A's and the SDS-930) record the type and number of system errors detected, total number of discrete transmissions, and the time duration of the test.

CONCLUSIONS

The simulator was successfully demonstrated in January 1969 with several RCA-110A test programs for the Instrument Unit (IU). The IU was used because it is the most complex system of the Saturn V, and it was determined that, if it could be simulated, then the other stages could be simulated also.

The checkout of the software for all timing and logic combinations was extremely difficult within a real-time computer network and the special hardware interface. Often hardware and software errors occurred simultaneously and obscured the source of the error.

The successful simulation runs of the IU test programs have proven that the concept of the real-time digital simulation for test program evaluation is feasible. Though the determination of all timing limitations with respect to size of data base and minimum component time-constants is still subject to further study, it is established that the simulator can be used in many applications without changes in the software. Systems that predominantly contain devices with relatively large time constants such as electro-mechanical, mechanical, and pneumatic devices are particularly suited for this simulator. Fast electronic logic circuits can also be simulated if they control other slower devices so that the minimum time between stimuli output and response input is in the range of milliseconds to seconds.

The software allows easy change of parameters, of tolerances, and of the configuration of the hardware under test for studying its effects on the overall systems performance, for evaluating the completeness of test programs, and for locating malfunctions. Hence, the real-time software simulator can be used for applications such as test-program design and evaluation, malfunction analysis, hardware design analysis, and training of checkout and launch personnel.

The data base can be set up easily since translation of the schematics into the logical/analog equations is not difficult. However, the data base should be established while the hardware systems are being designed so that the design engineers are modeling their own design, thus assuring the establishment of a data base that accurately reflects the hardware design. Also, the simulator can then be utilized as an active tool during the design phase and can be used for early test program design and evaluation before hardware delivery.

It is also conceivable that the logic / analog equations may describe the functions of subsystems on a higher level than the piece-part level, thus very large systems could be simulated to less depth.

Space missions of great complexity and of longer duration, and a greater frequency of launchings will require speeding up the checkout operations and a constant thorough knowledge of the status of all systems within the space vehicle. The vehicles themselves will be more complex and their configuration will be varied greatly, which will result in an increase in the design of new and more sophisticated test procedures. Narrow launch windows and more frequent launchings will result in the need for short turnaround time at the launch pad. To meet this challenge in new space programs such as the Space Shuttle and Space Station, plans are being made as to how this powerful simulation system can be optimally used for these new programs. Primarily, hardware changes for the interfaces have to be identified in order to make them more general. The software changes of the actual simulation program are expected to be minor; however, it might be necessary to write some data formatting routines for the interfaces.

REFERENCES

1. Saturn V System Development Breadboard Facility Data Plan. The Boeing Company, Document No. D5-15207, NASA Contract NAS8-5608, January 1965.

2. Saturn V System Development Breadboard Facility Operational Plan. The Boeing Company, Document No. D5-15201, NASA Contract NAS8-5608, November 1964.

3. ESE Simulation. General Electric Company, GE-Apollo Support Department, Daytona Beach, Fla., NASA Contract NASw-410, May 1965.

4. Jaegly, R. L.: Test Procedure Validation by Computer Simulation. AIAA Paper No. 69-280, AIAA 2, Paper Presented at Flight Test, Simulation and Support Conference, March 1968.

5. Brooks, L. W.; and Stahley, J. A.: Real-Time Digital Simulation of the Saturn V System. Sperry Rand Engineering Review, Systems-Computer Applications, 1969.

6. VLF-39-1 Digital Data Acquisition System (DDAS) for Saturn V. Interim Technical Report, MSFC-ASTR.

7. Brooks, L. W.; Gellman, L. J.; and Stahley, J. A.: Transfer Equation Preparation Manual for Launch Vehicle and Ground Support Equipment System Simulator. Prepared for NASA/MSFC/R-ASTR-ESA by Sperry Rand, Space Support Division, 1967.

8. Abe, R. G.; and Scarborough, K.: General Design Specification for Launch Vehicle and Ground Support Equipment Simulator System. Computer Sciences Corporation, March 1967.

REFERENCES (Concluded)

9. Program Specifications for Launch Vehicle and Ground Support Equipment Simulator System. Computer Sciences Corporation, February 1970.

10. Caldwell, S. H.: Switching Circuits and Logical Design. John Wiley & Sons, New York, N. Y., 1958, p. 468.

11. Balentine, T. L.: Operating Procedures for RCA-110A Computer Programs for DEE-6D Modification Acceptance Tests. Computer Sciences Corporation, June 20, 1968.

12. Acceptance Test Procedure — DEE-6 Simulator System. Scientific Data Systems, Document No. SDS-144934, NASA Contract NAS8-11809, May 1967.

13. Balentine, T. L.; Rigby, C. O.; and Abe, R. G.: Design Documentation for RCA-110A Computer Programs for DEE-6D Modification Acceptance Tests. Computer Sciences Corporation, December 19, 1967.

14. Brooks, L. W.; and McCubbins, C. L.: Integrated Test Plan for Acceptance of DEE-6D Modification. Sperry Rand Corporation, June 1967.

Page intentionally left blank

STATE VARIABLE DESCRIPTOR SYSTEM (SVDS)

By

N. F. Geer

SUMMARY

A digital computer program, State Variable Descriptor System (SVDS), has been developed for reducing the system block diagram of a dynamic system to a state variable description. Programs are also available that utilize the output of SVDS to perform particular types of analyses. The SVDS is intended to be used on systems with a number of time-invariant elements and a large number of integrators. However, the system being analyzed can contain time-varying and nonlinear elements. SVDS was designed for and implemented on remote time-sharing terminals and is intended for use by those persons familiar with state variable types of analyses. For a full utilization of SVDS, the user should have some knowledge of FORTRAN.

INTRODUCTION

A modern approach for analysis of linear systems is the state variable method. The state variable method permits describing a system in compact form by use of matrix equations. The matrix equations are generally in the form

$$\dot{X} = AX + BU \tag{1}$$

and

$$Y = CX + DU \quad , \tag{2}$$

where X is the state variable vector, U is the input vector, and Y is the output vector. A, B, C, and D are matrices that can be time varying. Techniques for manipulating these equations are covered in several texts [1, 2].

A general practice for finding the matrices for a system described by block diagrams is to reduce any transfer functions to simple integrators and summing junctions and assign the elements of the state vector as outputs of the simple integrators. The inputs to each integrator, expressed in terms of the state variables, form the A matrix, and the values of the output vector in terms of the state variables form the C matrix. The B and D matrices are formed from values of inputs to the integrators and the output vector in terms of the system input vector. For large complex systems, the manual determination of the matrices can become a long, tedious process. SVDS will determine the A, B, C, and D matrices and identify the elements of the vectors.

SVDS is particularly useful to those persons using state variables to describe large dynamic systems. The user can write time-sharing digital programs that utilize the outputs of SVDS to perform particular types of analyses. This report covers some of the types of analyses that have utilized SVDS.

MATHEMATICAL MODEL

The mathematical model generated by SVDS can best be visualized by the matrix block diagram of Figure 1. SVDS separates from the input those components that are linear time-invariant. These elements are summing junctions, gain constants, simple integrators, and transfer functions. Components that do not fall into these categories are grouped as nonlinear elements. The term nonlinear is used as a matter of convenience, and it should be understood that it includes linear time-varying elements as well as the usual nonlinear elements. In Figure 1, A, B, C, and D are constant matrices that SVDS determines from the linear time-variant elements. The N block of Figure 1 accounts for the nonlinear elements. N should not be interpreted as a matrix but as a group of functional relationships between elements of U* and Y*.

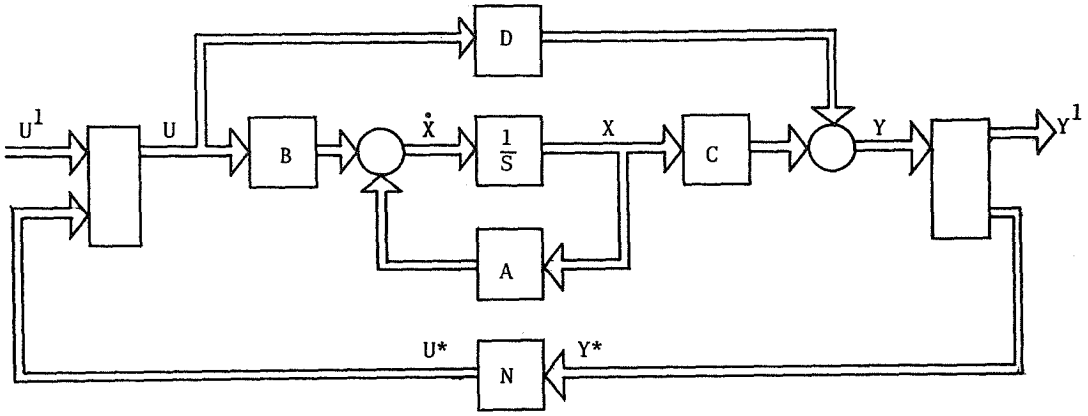


Figure 1. Matrix block diagram.

SYSTEM BLOCK DIAGRAM

The system to be analyzed uses five types or classes of blocks to describe the system. Table 1 lists these blocks. Because of the general block, class 5, these blocks are sufficient to describe any system. All the blocks are single output and are identified by a block number and a class number. The outputs of each block assume the name of the block (i.e., block number); thus, there must be no duplication of block numbers. The following describe each class of blocks:

1. Class 1 — Input Block

Class 1 designates the input points to the system. Class 1 blocks are used to form the U^1 vector.

2. Class 2 — Summing Junctions

Summing junctions also include gain constants. If a summing junction has only a single input, it then becomes a simple gain.

3. Class 3 — Simple Integrator

Class 3 is used to designate simple integrators in the system. The outputs of these simple integrators form the state vector.

4. Class 4 — Transfer Functions

Class 4 is for transfer functions that are the ratio of two polynomials in the S-domain. The only restriction is that they must be realizable transfer functions; i.e., the order or the numerator $\geq$ the order of the denominator. The transfer functions are reduced by SVDS to simple integrators and summing junctions. The reduction method used is the so-called "M-Method."

For the M-Method of reduction, the transfer function is placed in the following form:

$$Y = \frac{\left(A_n S^n + A_{n-1} S^{n-1} \ldots\ldots + A_1 S + A_0\right)}{\left(S^n + B_{n-1} S^{n-1} \ldots\ldots + B_1 + B_0\right)} U .$$

The simulation diagram is as follows:

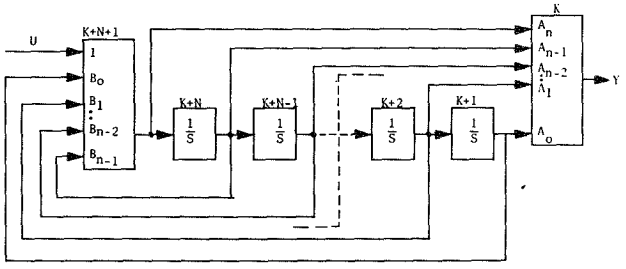
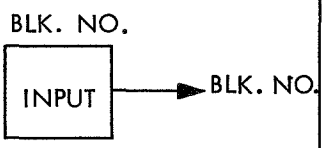
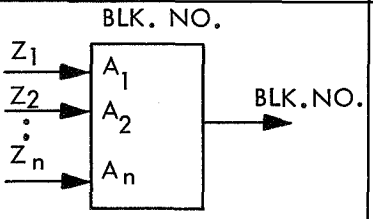
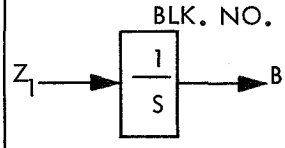
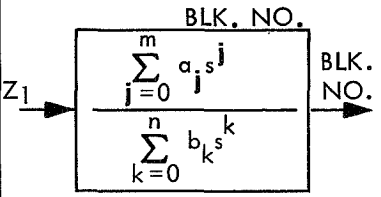
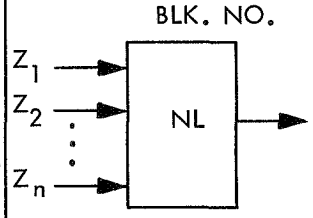


TABLE 1. CODING OF BLOCK DIAGRAMS FOR COMPUTER INPUT

CLASS	DESCRIPTION	INPUT FORMAT
1 (INPUT)		BLK. NO., 1
2 (SUMMER)		BLK. NO., 2 $n, Z_1, A_1, Z_2, A_2, \ldots . Z_n, A_n$
3 (INTE- GRATOR)		BLK. NO., 3 Z_1
4 (TRANSFER FUNCTIONS)		BLK. NO., 4 m, n, Z_1 $a_0, a_1, \ldots . a_m$ $b_0, b_1, \ldots . b_n$
5 (NON-LIN. OR TIME VAR. FUNCTIONS)		BLK. NO., 5 n $Z_1, Z_2, \ldots . Z_n$

SVDS generates the additional blocks, assigns the transfer function block number to the output summing junction, and assigns consecutive numbers to the other blocks starting with the integrator nearest the output summing junction. Thus, for transfer functions, N + 1 additional consecutive block numbers must be reserved where N is the order of the denominator. For instance, if a transfer function is of order 3 and has a block number of 5, block numbers 6, 7, 8, and 9 will also be used.

5. Class 5 — Nonlinear and/or Time-Varying Blocks

The class 5 blocks are general and are used to designate functions not covered by the

other classes of blocks. When these blocks are encountered, the system input vector is augmented by the U* elements. The output vector is searched for the required Y* elements. If they are not present in the Y vector, the Y vector is augmented to accommodate the Y* elements. SVDS maintains a record of these blocks for identification purposes.

6. Output Vector

The output vector is initially established by listing in the input to SVDS those blocks whose output is to make up the system output vector. As stated above, the class 5 block will augment this vector if required.

PROGRAM OPERATION

The program is divided into three major routines: (1) input routine, (2) variable identification routine, and (3) matrix generation routine. The following describes the operation of each of these routines. A description is also included of the routine that generates a set of simulation equations. The sample problem of Figure 2 will be used to illustrate outputs for the various routines.

Input Routine

The first item of input is the system output vector description Y^1. This is performed by entering the number of output points and the list of block numbers whose output makes up the output vector. The remainder of the block diagram information can be entered in any order. Encoding of the blocks is that shown in Table 1. The first two items of each block are the block number and class. The block class is used as a control, not only to catalog the block, but also to read the remainder of the block information.

The input routine performs certain error checks. As each block is read, the number is checked against previous blocks for a duplication of block number. If a duplication does occur, the block information is not processed and a program abort will occur on completion of the reading of input. The input routine also checks the transfer functions, class 4, for realizability. Nonrealizable transfer functions will also cause an abort.

The input routine prints out any missing blocks and redundant blocks. When this occurs, the program will continue, but the user is alerted to possible errors in the input. If a block requires input from a missing block, the output of the missing block is assumed to be zero.

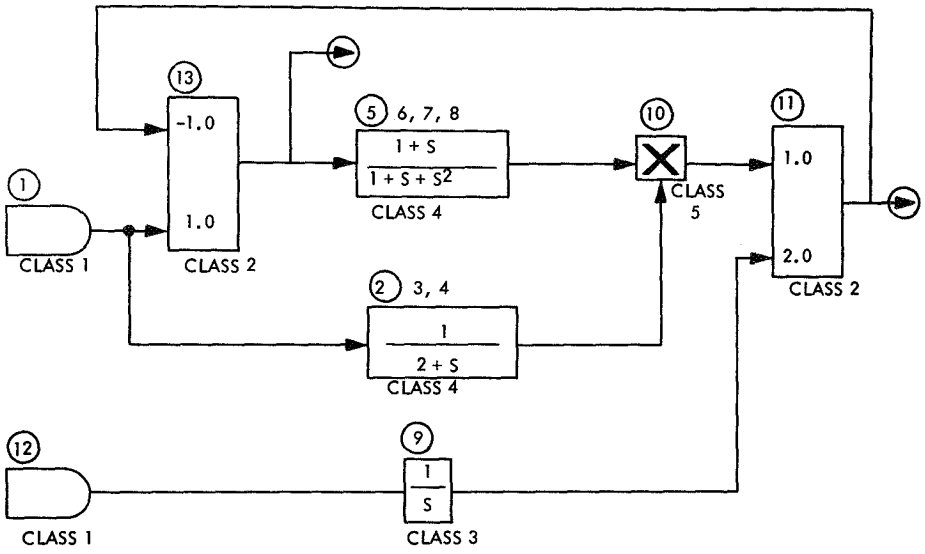


Figure 2. Sample system block diagram.

Variable Identification Routine

The variable identification routine identifies the elements of the X, U, and Y vectors to the outputs of the specific blocks of the block diagram. The state variable vector elements, X_i, are assigned to outputs of the simple integrators including those of the transfer functions. The input vector elements, U_i, are assigned to the outputs of the system input blocks, class 1, and the outputs of the class 5 blocks. The output vector elements, Y_i, are assigned to the initial blocks listed for output and augmented by class 5 blocks when required. In all cases, the sequence for assigning the indexes is the sequence encountered on input. In addition to these vector identifications, identification of the class 5 elements is provided. The following is an identification list for the system on Figure 2.

VARIABLE IDENTIFICATION							
X	BLOCK	X	BLOCK	X	BLOCK	X	BLOCK
1	3	2	6	3	7	4	9
Y	BLOCK	Y	BLOCK	Y	BLOCK	Y	BLOCK
1	11	2	13	3	5	4	2
U	BLOCK	U	BLOCK	U	BLOCK	U	BLOCK
1	1	2	10	3	12		

The following functions are to be supplied by the user:

OUTPUT		INPUTS			
U BLOCK		Y BLOCK		Y BLOCK	
2	10	3	5	4	2

Matrix Generation Routine

After the input routine is completed, the block diagram is retained in memory as four arrays. These arrays are from the first three classes of blocks and the output vector. Transfer function blocks, class 4, no longer exist since they have been reduced to simple integrators and summing junctions. The nonlinear elements, class 5, have been replaced by augmenting the input and output vectors. However, an identification list for nonlinear elements is maintained in the disk files. Thus, after the input routine is completed, the only thing contained in memory is the block description of the linear time-invariant portion of the system. The A, B, C and D matrices are generated from the linear time-invariant elements of the system.

From equations (1) and (2) it is seen that the A and C matrices are coefficients of the state vector, and the B and D matrices are coefficients of the Y vector. To compute these matrices one can think of the matrices as being the gains between particular points in the system. As an example, the first column of the A matrix contains the gains between the output of the integrator that has been designated as x_1 and the inputs to each integrator. In other words, if the output of x_1 is assigned the numerical value of 1 and the inputs to each integrator are evaluated, these values will be the elements of the first column of the A matrix. It should be noted that these values are a result of the gain constants contained in the summing junction, class 2, blocks only. Columns of the C matrix are found from the values of the output points. The B and D matrices can be found in a similar manner, except the output of the system input blocks are set equal to 1.

The procedure used by SVDS to accomplish this is to establish an output array for the blocks. Indexing of the array is by the block number, and the output of each block can be computed and stored in this array. For the J column of the A and C matrix, the block output array is nulled, and the output array element that corresponds to the integrator for x_J is set equal to one. Since the interest is only in the gain between x_J and the inputs to the integrators and the system output points, only the output of the summing junctions requires evaluation. To obtain this evaluation it is necessary to iterate the summing junctions. The reason for this iteration is that the description of the summing junctions are stored in a random manner in the summing junction array. For instance, if summing junction N requires input from summing junction M and N is stored before M, N will not obtain the correct input until the second evaluation. The iterations are repeated until there is no change in the outputs of the summing junctions. When the outputs of the summing junctions have converged, the numerical value of the inputs to integrators will form the J column of the A matrix, and the value of the system output vector is the J column of the

C matrix. These values are written directly into an A matrix disk file and a B matrix file. Thus, no additional computer memory is required to store these matrices.

The B and D matrix files are determined in the same manner except the outputs of the system input blocks, class 1, are set equal to 1. The first two items of information contained in all matrix files are the dimensions of the matrices. These dimensions are determined from the variable identification routine. When a null matrix occurs, the dimensions are set to 0, 0.

In the present system no provision is made for solving linear algebraic loops. If loops do occur, the iteration of the summing junctions will fail and the program will abort. Algebraic loops are not a frequent occurrence, but it is recognized that they can exist. Methods for solving algebraic loops are under evaluation.

State Equation Routine

The state equation routine generates a subroutine of FORTRAN statements to be used in system simulation. These FORTRAN statements contain all the information contained in the matrix equations but are in a more compact form. In general, the matrices will contain a very large number of zeroes, but in the state equation routine, terms with zero coefficients are not printed.

From Figure 1 the simulation equations in matrix form are:

$$U^1 = F(t) \quad ,$$

$$Y = CX + DU \quad ,$$

$$U^* = N(Y) \quad ,$$

and

$$\dot{X} = AX + BU \quad .$$

If the D matrix exists, it can be seen that the U vector depends on Y, and Y depends on the U vector. Since digital computations are sequential, the equations for the U* and Y elements must be properly sequenced.

The method used is to establish two Boolean type matrices; i.e., 0, 1 matrices. The first matrix, E, is obtained from the nonlinear identification and indicates the required elements of the Y vector for each U element. The second matrix, DP, is obtained from the D matrix by replacing non-zero elements with a 1. The matrices for the sample problem would appear as follow:

E MATRIX

		Y 1	2	3	4
	1	0	0	0	0
U	2	0	0	1	1
	3	0	0	0	0

DP MATRIX

		U 1	2	3
	1	0	1	0
Y	2	1	1	0
	3	0	0	0
	4	0	0	0

The zero rows in the E matrix indicate that those U elements are external inputs to the system. The zero rows of the DP matrix indicate that those Y elements are functions only of the state variables.

Sequencing is started by searching the E matrix for a zero row. As each zero row is encountered, the U index is stored in a sequencing array, and the corresponding column of the DP matrix is set to zero. All the rows found on the first search of the E matrix will be the external forcing functions. After the search of the E matrix is complete, the DP matrix is searched for zero rows. As each zero row is encountered, the Y index is stored in the same sequencing array used by the U vector, and the corresponding column of the E matrix is set at zero. The process is repeated until no zero rows are found. The stored indexes in the sequencing array are in the proper computation sequence, and this index array is used by the remainder of the routine to control the printing of the U_i and Y_i equations.

Although it has not occurred in actual practice, it is possible that an algebraic loop can exist through the nonlinear elements and the D matrix. If this should occur, the elements cannot be sequenced by the above procedure, and the routine will indicate those equations that cannot be sequenced.

The remainder of the routine is primarily a print routine. The files of the A, B, C, and D

matrices are used to determine the coefficients. Any zero coefficient terms are not printed. Figure 3 gives the printout for the example of Figure 2.

```
SUBROUTINE DERIV
COMMON T,XDOT(4),X(4),Y(4),U(3)
U(1)=EXTERNAL
U(3)=EXTERNAL
Y(3)=X(2)+X(3)
Y(4)=X(1)
U(2)=FUNCTION OF Y'S 3 4
Y(1)=2.00000E+00*X(4)+U(2)
Y(2)=2.00000E+00*X(4)+U(1)-U(2)
XDOT(1)=-2.00000E+00*X(1)+U(1)
XDOT(2)=X(3)
XDOT(3)=-X(2)-X(3)-2.00000E+00*X(4)+U(1)-U(2)
XDOT(4)=U(3)
RETURN
END
```

Figure 3. Subroutine example.

APPLICATION

Figure 4 shows the role that SVDS has played in various types of analyses. The primary outputs of SVDS are A, B, C, and D matrices and the nonlinear identification. This information is stored on disk files, and particular types of analysis use these files.

Simulation (Time Response)

For system simulation the subroutine DERIV described previously is used by available library numerical integration routines to obtain the system time response. Before compiling the DERIV subroutine, the user must supply the specific functions for the elements of the U vector. U(i)'s that are designated as EXTERNAL are the system forcing functions. Each must be replaced by specific FORTRAN statements. These external forcing functions are, in general, functions of time. In studies on optimal switching functions, they can be functions of the state variables.

The U(i)'s that are generated by the class 5 blocks must also be replaced by specific FORTRAN statements. The required Y(J)'s to generate the U(i) are specified in each case, and it is only necessary for the user to supply the relationship. It is evident that forcing functions and nonlinearities could be automatically generated. In the development of SVDS, consideration was given to providing these functions. However, it became apparent that for time-sharing computer users, this did not serve a useful purpose but tended to restrict the general application of SVDS. If the various types of nonlinearities and forcing functions were provided automatically, it would be necessary to have a large catalogue of these functions. However, in particular applications, only a small portion of the catalogue would be used, but the total catalogue must be

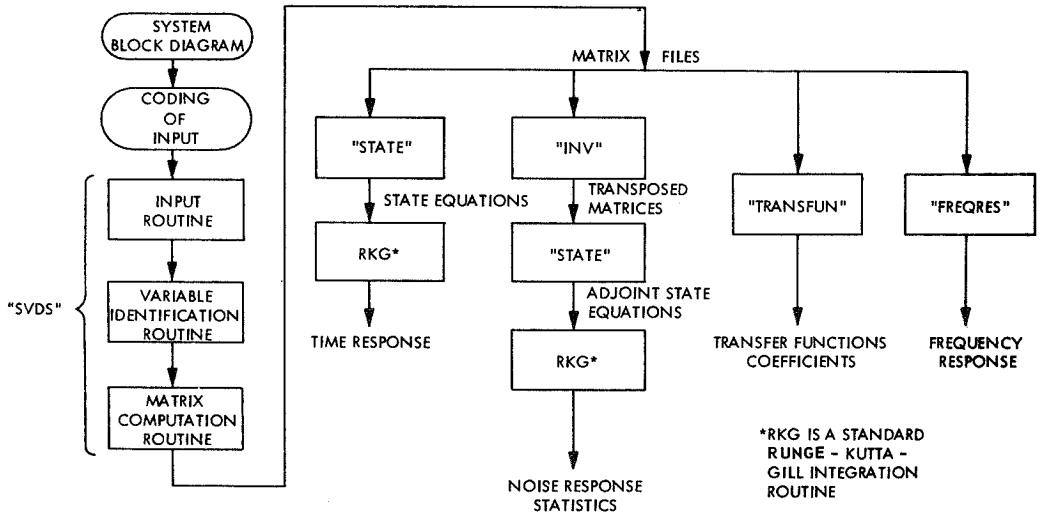


Figure 4. Role of SVDS in dynamic system analysis.

available for use as subroutines. In time-sharing applications, computer memory is usually at a premium and to include a comprehensive catalogue of functions would require extensive file operations resulting in slower program operation. Many non-linear functions are very simple FORTRAN statements. By having the user supply the FORTRAN statements, the user is not restricted to functions available in a catalogue, but by his own ability to describe the function. In time-sharing remote terminal operation, this user capability is very practical.

There are usually several types of integration routines available in a time-sharing library. A common routine is the fourth order Runge-Kutta. Runge-Kutta is particularly suited to remote terminal operation, because its accuracy can be quickly checked by integrating over a small time interval and then comparing results obtained over the same time interval but using one-half the original integration interval. It has been this writer's experience that Runge-Kutta is faster than predictor-corrector methods when the system contains discontinuous functions. Figure 5 gives a typical flow diagram for carrying out the numerical integration.

Adjoint Systems

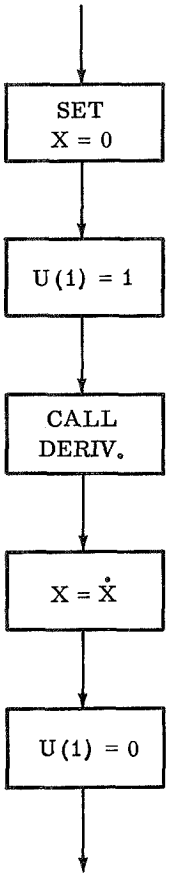
For time-varying systems it is sometimes desired to find the impulse response matrix, $H(\tau, T)$, where τ represents the time of the impulse input to the system and T is the time of output. H is also referred to as a weighting function. Very often, the output at a specific time T_1 resulting from inputs at specific times in the past is desired. In a linear time-invariant case, the response at T_1 depends only on the time difference, $T_1-\tau$. In time-varying cases the response is dependent on the specific time of input. To find the response at T_1 resulting from impulse inputs at specific τ's by use of normal state equations would require an integration for each τ. The reason for this is that the independent variable or running variable is T and not τ. In an adjoint system only one integration is required for a specified terminal time T_1. For the adjoint system, τ becomes the running variable with real time running backwards. Essentially, the adjoint of a system is found by interchanging inputs and outputs of each block on the block diagram. When this is performed, the Y^1 vector becomes an input vector, U^1 becomes an output vector, summing junctions become branch points, and branch points become summing junctions. Thus, one could obtain the adjoint of a system by redrawing the system block diagram and executing SVDS. However, there are direct relationships between the matrix descriptions of a system and its adjoint. The adjoint of Figure 1 would be that shown in Figure 6.

One can now note that the form of Figure 6 is the same as that of Figure 1. The difference is that the matrices have been transposed, and the B and C matrices interchanged. The N block contains all the time varying constants and, in this case, can be treated as a matrix.

The routine INV of Figure 4 reorganizes the file names so that the routine STATE will read the correct files to give the adjoint equations. The DERIV routine will contain the same variables as for the forward system; i.e., U is the input vector to the adjoint system and Y is the output vector. However, the indexes of U and Y are interchanged. The circled variables of Figure 5 are the variables used by DERIV.

To obtain the impulse response matrix at T_1, impulses must be used as inputs, in sequence, on the input elements. Most often Y^1 is a scalar and the output U^1 can be a vector. Integration is carried out with T running from zero to T_1 and the time varying coefficients computed from the real time, T_1-T.

For simulation of a unit impulse, the equivalence of initial conditions on an integrator and an impulse input is utilized. If the input to an integrator is a unit impulse through a gain K, this is equivalent to an integrator with zero input and an initial condition of K. The simulation flow diagram of Figure 5 is the same except the initialization of the X vector. For a scalar input, U(1), this block is replaced by the following operations:



Transfer Functions

The routine TRANSFUN was developed for the purpose of finding the transfer functions for passive linear networks. Theoretically, it can be applied to any linear time-invariant system and obtain the transfer functions between input points and output points of the system. However, it should be pointed out that transfer functions do not contain any more information than is contained in the state variable description of the system. As an example, the poles of a system are the characteristic values of the A matrix. In reality, the problem that is being solved by TRANSFUN is the changing of the state variable description of a linear time-invariant system to the classical transfer function description.

For a linear time-invariant system described by

$$\dot{X} = AX + BU \qquad (3)$$

and

$$Y = CX + DU \quad , \qquad (4)$$

the transfer function description would be

$$\overline{Y} = G(s)\,\overline{U} \quad ,$$

where $\overline{Y}$ and $\overline{U}$ are the Laplace transforms of Y and U. $G(s)$ is the transfer function matrix with elements $g(s)_{J,K}$. Each element has the form

$$g(s)_{J,K} = \frac{a_0 + a_1 s + a_2 s^2 + \ldots . a_p s^p}{b_0 + b_1 s + b_2 s^2 + \ldots . b_n s^n}$$

with $p \leq n$. $G(s)$ is found by taking the Laplace transform of equation (3) and solving for $\overline{X}$ to give

$$\overline{X} = [sI - A]^{-1} B\overline{U} \quad . \qquad (5)$$

Taking the Laplace transform of equation (4) and substituting for $\overline{X}$ will give

$$\overline{Y} = [CEB + D]\,\overline{U} \quad ,$$

and, thus,

$$G(s) = CEB + D \quad ,$$

where

$$E = [sI - A]^{-1} \quad .$$

TRANSFUN performs the inversion of $[sI - A]^{-1}$ using a method similar to that used by Zadeh [1].

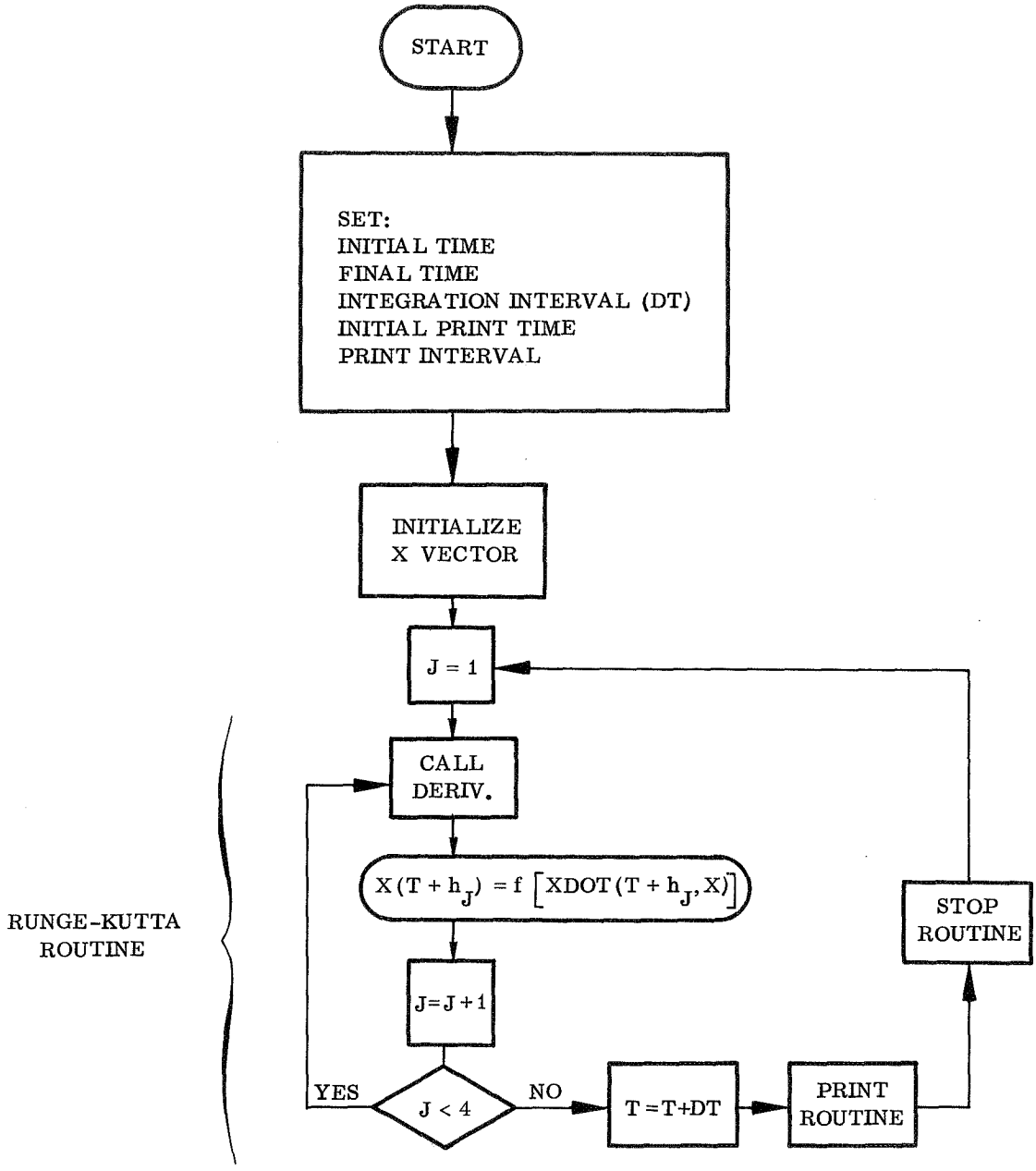


Figure 5. Numerical integration by the Runge-Kutta method.

Each element of the transfer function matrix is the ratio of two polynomials in s. The denominator of each element is the same for all elements and is the characteristic equation obtained from the A matrix. After the coefficients of the numerator and denominator are found, a root finder routine can be used to find the poles and zeroes of the system. In many cases reduction of the transfer function can be achieved by the cancellation of identical poles and zeroes.

Frequency Response

FREQRES finds the frequency response between an input point, specified by the user, and specified

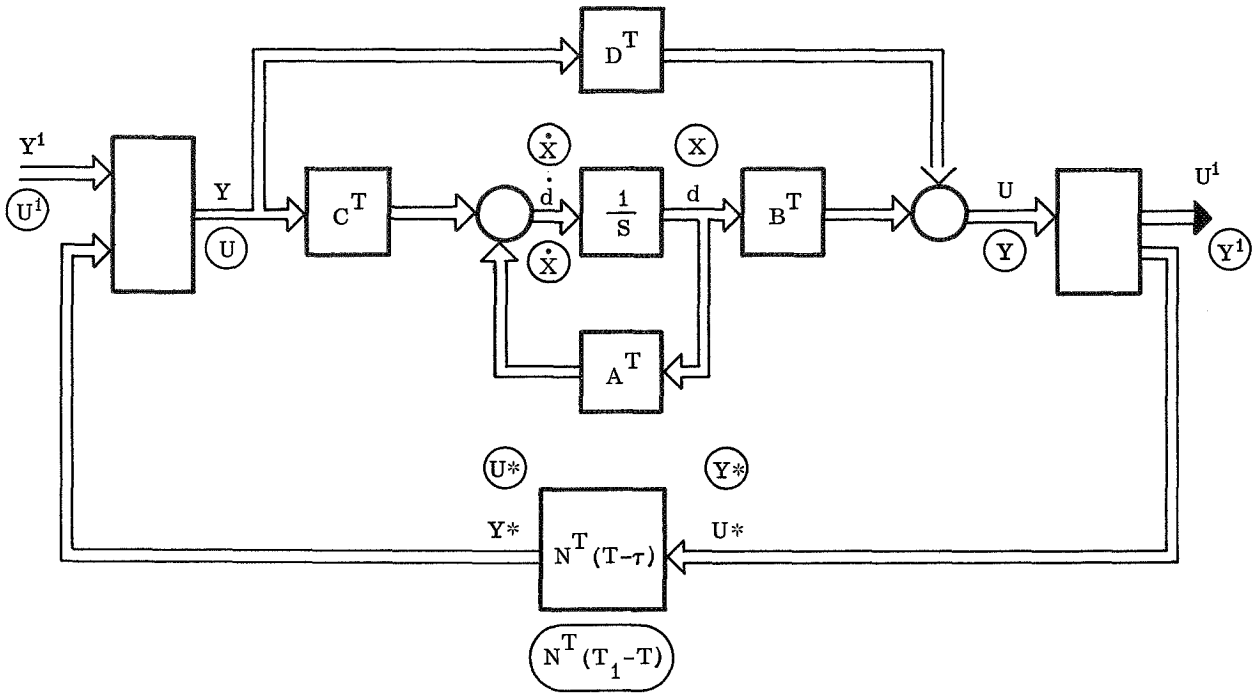


Figure 6. Adjoint system.

output points for the linear time-invariant portion of a system. The frequency is in Hertz, and the response is given in both rectangular and polar form. The use of FREQRES is recommended only on systems of tenth order or less. The routine requires the inversion of complex matrices, and, for large matrices, the computation time can be very long. The computation method is made simple by the use of complex variables and complex matrix routines.

For the linear time-invariant portion of the system the state equations are, as given before:

$$\dot{X} = AX + BU \tag{6}$$

and

$$Y + CX + DU \quad . \tag{7}$$

In the frequency domain these can be written as

$$j\omega \tilde{X} = A \tilde{X} + BU \tag{8}$$

and

$$\tilde{Y} = C\tilde{X} + DU \quad , \tag{9}$$

where $\tilde{X}$ and $\tilde{Y}$ are complex and $j = \sqrt{-1}$. From equation (8)

$$\tilde{X} = [j\omega I - A]^{-1} BU \quad ,$$

and

$$Y = C[j\omega I - A]^{-1} BU + DU \quad .$$

At each frequency step, $[j\omega I - A]^{-1}$ is evaluated, and, if A is large, this can be a long computation process. After this inversion is found, the remainder of the computation is by complex matrix multiplication and addition. In the computation, only a selected input element in the U vector is set equal to one and all other elements are set equal to zero.

CONCLUSIONS

For the analysis of large complex systems, SVDS provides the engineer with a fast method for obtaining a state variable description of the system. The SVDS is intended for remote terminal operation. All computation routines are maintained in BCD form and can be changed, if required, by the user. At the present time, the SVDS is implemented on the GE 605 Desk Side computer system. An 1108 version is being developed.

REFERENCES

1. Zadeh, L. A.; and Desoer, C.A.: Linear System Theory. The State Variable Approach, McGraw-Hill, New York, N. Y., 1963.

2. Derusso, P. M.; Roy, R. J.; and Close, C. M.: State Variables for Engineers. John Wiley & Sons, Inc., New York, N. Y., 1965.

STORAGE SCOPE GRAPHICS AND EXPERIMENTAL APPLICATIONS

By

R. Seitz

INTRODUCTION

One of the research tasks that the Man-Machine Systems Branch of the Computation Laboratory at Marshall Space Flight Center has undertaken is the investigation of low priced graphic display equipment. At the present time, most interactive graphic terminals cost between $ 50 000 and $ 100 000 per station. This Branch has been investigating a new type of graphic display unit that costs between $ 5000 and $ 10 000 per station, putting it in the price range of a keypunch or an 1108 teletype. These low priced graphic displays are not a direct substitute for the more expensive graphic displays but should suffice for a number of applications within NASA such as computer-aided design, engineering simulation, and management information display. The display consoles under investigation utilize one or two Tektronix 611 storage display scopes, a pushbutton keyboard, and an electronic interface. They may or may not include a selectric typewriter, a graphical input device, and, when it becomes available, a $ 3000 photostatic copier. Another option that may be available soon is a high speed acoustic telephone coupler. This would permit the Tektronix terminal to communicate with a computer through any standard telephone receiver, thereby rendering the graphics station truly portable. Figure 1 shows a prototype version of such a display unit. This particular unit is interfaced to an IBM 1130 computer and has been operational since early 1968. It affords a character-writing rate of about 900 characters per second.

Figure 2 shows an application for which one of these terminals is currently being used; i.e., the design and display of hypersonic lifting bodies similar to the space shuttle configurations. The IBM 1130 computer, with display, is used interactively to assist the designer by filling in the airframe in detail and displaying it, based upon general design information supplied by the user. When a design appears to be acceptable, its X, Y, Z coordinates are punched out on cards for simulation on the UNIVAC 1108's.

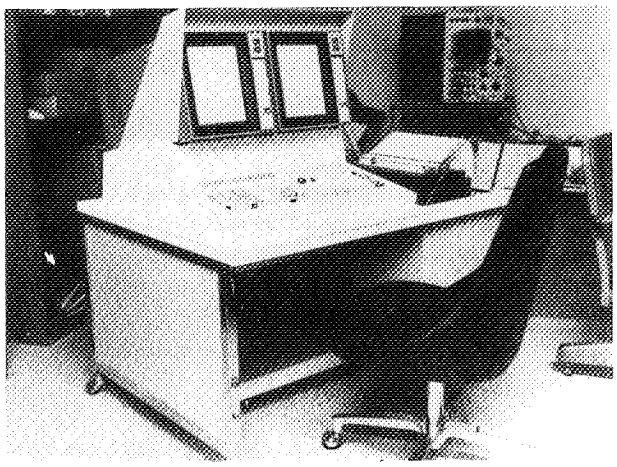


Figure 1. Prototype version of AMTRAN terminal.

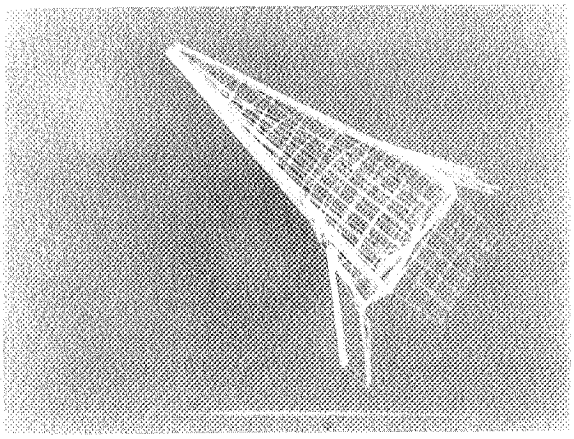


Figure 2. Space shuttle configuration.

Figure 3 shows one of the circuits used in constructing the interface between the low priced display and the IBM 1130. This suggests another application for these low priced displays; i.e., printed circuit board design. It is noted that this figure was prepared from a Polaroid transparency

taken directly from the scope face. Polaroid transparencies may be used to prepare "instant slides", with the picture generated and formatted by the computer.

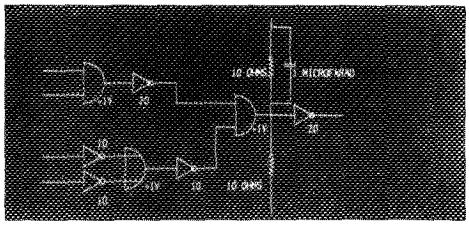


Figure 3. Logic circuit.

Figure 4 shows a printed circuit board layout prepared as a demonstration of the resolution and information content of the scopes.

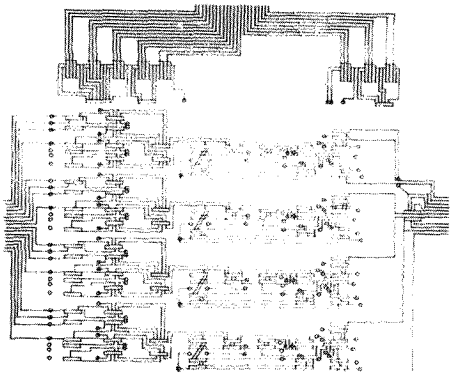


Figure 4. Printed circuit board.

Figure 5 shows a plot of the Fresnel integral $\int \sin\left(\frac{X^2}{2}\right) dx$.

Figure 6 shows a family of solutions of a differential equation, illustrating some scientific applications of such displays.

Figure 7 shows a Form 1125 (Request for Computational Support) as it appears on the storage scope display. This figure exemplifies another application well-suited to the storage-scope displays; editing of the text appearing on forms. Graphic capability is required to display the form, and high resolution is necessary to display the fine print and special character sets found on forms. Also, the low cost Tektronix photostatic copier is desirable to provide a hard-copy record of alterations to the form.

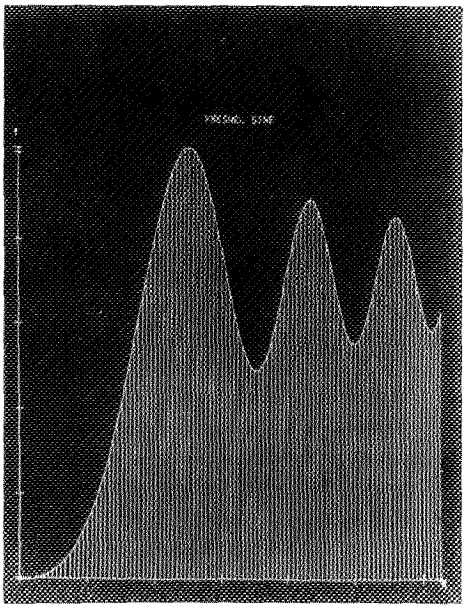


Figure 5. Fresnel curve.

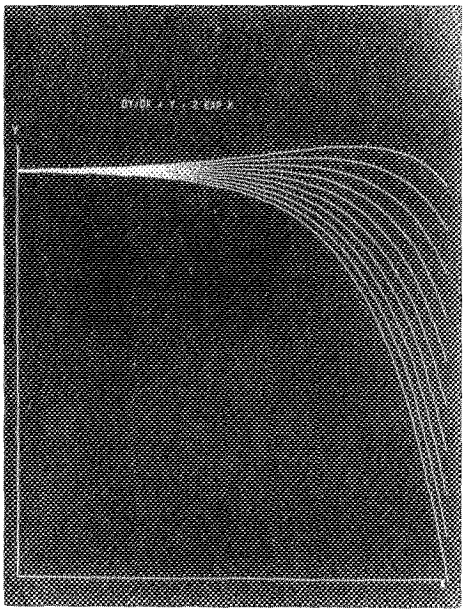


Figure 6. Solution of a family of differential equations.

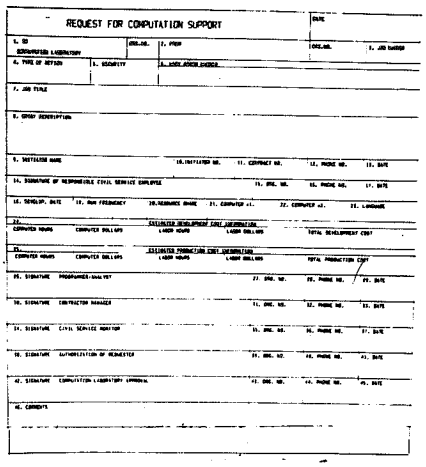


Figure 7. Computation Laboratory Form 1125.

Figure 8 shows an automatically formatted schedule chart for one of the projects of the Man-Machine Systems Branch. Such schedule charts may be updated on a weekly basis. This chart was automatically formatted and taken directly from the scope using a special Polaroid camera marketed by Tektronix.

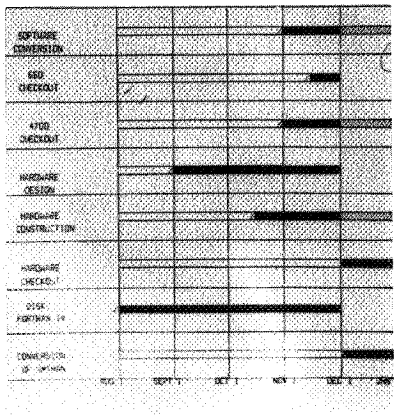


Figure 8. Schedule for branch project.

The Man-Machine Systems Branch has developed software and has encouraged the development of hardware for storage-type graphics displays. They are now felt to be ready for use at the Marshall Space Flight Center and at other NASA installations.

TEXT AND FORM EDITING

One application utilizing the low cost graphics terminal has been the development of an experimental Text and Form Editing program. An objective of this study has been to develop general, but simple, software in an effort to define and evaluate how man can interact with the computer, and to determine the value of such capability in a management information environment. The development of a prototype system on the IBM 1130 computer is being coordinated with the Research Planning Office at MSFC. The system will provide online updating and editing capabilities for a file of approximately 500 Research and Technology Resumes, NASA Form 1122. The system currently provides the capability to allow for definition of up to 20 different single-page business forms and to establish files of associated information. The system provides software that allows online creation, maintenance, editing, and display of text information in the form. The program utilizes a two-scope display unit. The left scope is used for displaying the form and current information. The right scope is used to display commands and to provide a working area for item construction and editing.

The experimental system has been programmed to recognize 16 mnemonic commands to facilitate the display, editing, and maintenance processes. In addition to the commands, the system recognizes a character set consisting of 54 alphabetic, numeric, and special characters. Commands are entered by pressing the operator keys followed by alphanumeric file, code, and item identifiers. The character keys are arranged in standard typewriter position and are used to type in identifying information following operators and for entering text information. Commands and characters are immediately displayed on the scope upon entry.

The following commands are recognized by the system:

1. FILE — Identifies a particular file of information and the associated graphic form.

2. FORM — Causes the graphic form associated with the previously identified file to be displayed.

3. CODE — Causes the specified logical record of information to be read from the random access disk into an in-core work area. When the information is read, the form and associated information is displayed on the scope.

4. ALL — Causes all current information from the in-core work area to be displayed in the appropriate blocks on the form.

5. EXCEPT — Causes all information except specified items to be displayed in the appropriate blocks on the form.

6. LIST — Causes specified items to be displayed in the appropriate blocks on the form.

7. ENTER — Allows information for a specified item to be input and concurrently displayed in the construction area on the right scope.

8. EDIT — Initiates the editing process for a specified item by displaying the current contents of the item at the top of the construction scope. The EDIT feature of the system is under cursor control, and the following control keys operate in conjunction with the EDIT command:

a. LINE FEED

b. WORD FEED

c. CHARACTER FEED

d. LINE DELETE

e. WORD DELETE

f. CHARACTER DELETE

g. CONTINUE

Using the feed controls keys, the user steps through an item down to the position at which he wishes to modify the text. As the user steps through the item, a second copy of the information is displayed in the middle section of the construction scope. The cursor indicates the current position in the original item at the top of the scope. When the position at which a change is to be made is reached, the user can strike out information in the original by using the delete control keys. New information can then be typed in to replace the deleted information. At this point, the cursor will move to the second copy in the middle of the scope and will follow along as new information is entered. The stepping and deleting process can be continued by pressing CONTINUE, if subsequent information in the item is to be edited. Strict insertion is accomplished by stepping to the position and typing in information without deleting. The EDIT procedure is completed by pressing the EOM key. Upon completion of an EDIT, the contents of the item are displayed at the bottom of the construction area for final verification. The edit algorithm provides automatic line length adjustment so that information will fit in the block provided on the graphic form.

9. TRANSFER — Causes the information resulting from an ENTER or EDIT operation to be transferred into the in-core work area, replacing the previous information for the item.

10. ABORT — Causes a command to be aborted without modifying the in-core information.

11. ERASE — Causes the left display of the form and information to be erased.

12. PRINT — Causes the current in-core information to be printed on the 1132 Printer along with the identifying titles for each item number.

13. TYPE — Causes the current in-core information to be typed on a NASA 1122 form utilizing the IBM selectric typewriter on the console.

14. STORE — Causes all of the current in-core information associated with a particular work order number to be stored permanently on the disk in place of the old information.

15. NEW — Causes the system to be set up for the specified new code number. The form is displayed on the left scope, and the system is ready to start accepting information using the ENTER command.

16. DELETE — Causes the specified code number to be deleted from the disk file.

Example of An Edit Operation

Figures 9 through 12 are provided to illustrate an example of the editing procedure. This process is the result of the following operations:

1. FILE 1122 — Identifies the file room from which information is to be processed.

2. FORM — Causes the form (Figure 9) to be displayed on the left scope.

3. CODE 125-23-02-20-62 — Causes a logical record of information associated with the work order number to be read from the disk and displayed on the left scope along with the form (Figure 10).

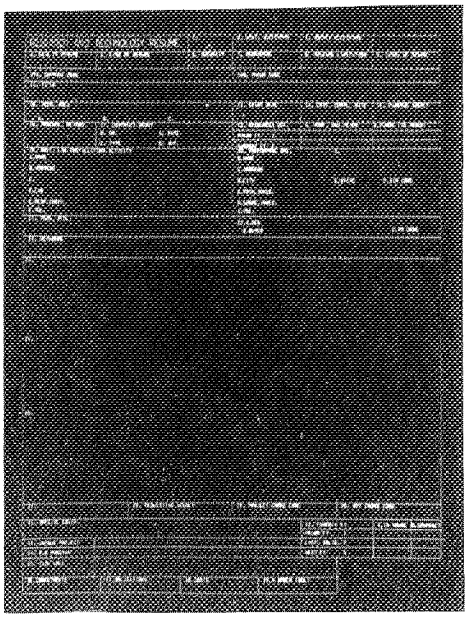


Figure 9. Form 1122.

4. EDIT 24 — Causes the contents of item 24 to be displayed at the top of the right scope (Fig. 11). Figure 11 shows a string of X's and Y's replacing the word THE on the third line of the item.

5. TRANSFER — Causes the modified item to replace the in-core information for that item.

6. ERASE — Causes the left scope to be erased.

7. FORM — Causes the blank form (Fig. 9) to be displayed on the left scope.

8. ALL — Causes the information to be displayed in the form (Fig. 12). Note the change in item 24.

9. The system is ready for the next command.

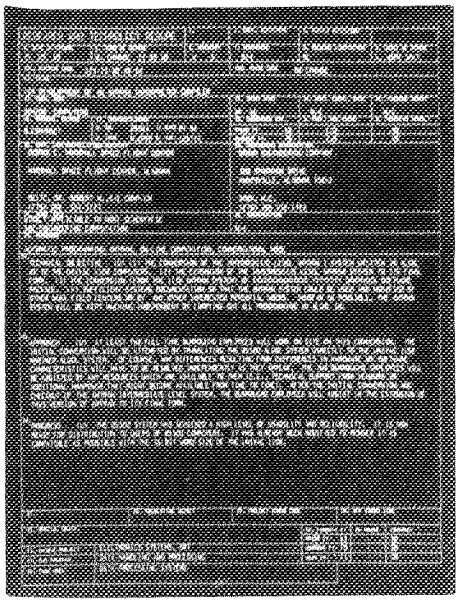


Figure 10. Completed Form 1122.

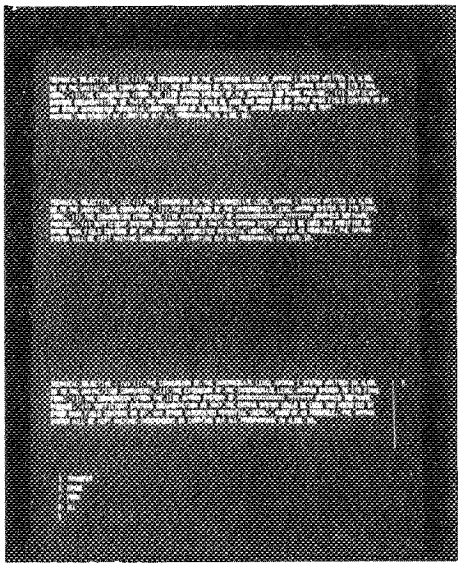


Figure 11. Editing process.

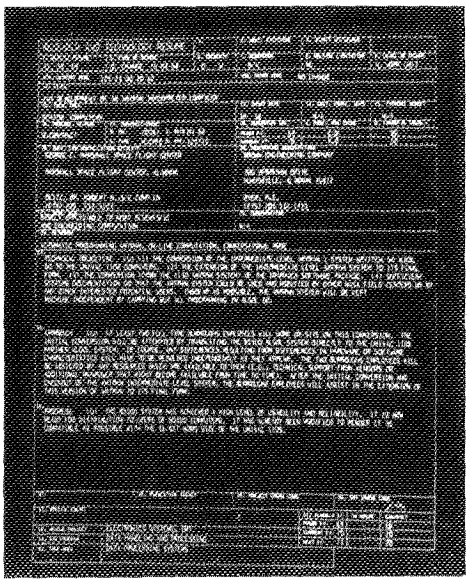


Figure 12. Edited process displayed on 1122.

AMTRAN

Automatic Mathematical TRANslation, is a conversational mode, mathematically oriented language developed to assist engineers and scientists in solving a wide variety of mathematical, statistical, and engineering problems with a minimum of programming effort by the user. Table 1 shows the current status of the various implementations of AMTRAN.

The motivation for developing AMTRAN was to reduce programming costs and improve convenience and accessibility conditions for the user. The total programming costs throughout NASA probably exceed 100 million dollars per year. Any techniques that could significantly reduce that expenditure rate would quickly pay for their development costs.

TABLE 1. CURRENT STATUS OF THE IMPLEMENTATION OF AMTRAN

Machine	Implementation Language	Remarks
IBM 1620	MACHINE	· Completed in 1966— Obsolete
UNIVAC 1108	ALGOL	· Usable on B5500 · Not Usable on 1108 · Extended Programming Features · Time-Shares 10 Teletypes · Somewhat Machine-Dependent
IBM 1130	FORTRAN IV & MACHINE	· Ready for User Evaluation · Runs on Standard MSFC 1130 · Slow, Basic, but Reliable
IBM 1130	MACHINE	· University of Georgia · Fast, New, Untested · Runs on Standard MSFC 1130

Some of the features of AMTRAN which distinguished it from FORTRAN are:

1. Automatic vector and matrix arithmetic are provided, together with vector and matrix operators such as SUM, TRANSPOSE, CONCATENATE, etc. This reduces the need for constructing DO loops when carrying out operations on one- and two-dimensional arrays.[1]

2. Less manual bookkeeping is required than is the case with FORTRAN. Dynamic memory allocation eliminates the need for DIMENSION and disk calls. Other similar features exist that tend to free the programmer from a certain amount of detailed accounting.

1. AMTRAN's array arithmetic contributes two important side benefits in addition to reducing the programming requirements. AMTRAN utilizes an interpreter to provide special services such as error checks and dynamic memory allocation to the user during the execution of a program. In general, interpreters are very slow and inefficient in execution compared to FORTRAN compilers. However, AMTRAN's automatic array arithmetic tends to lead to execution speeds that approach the speeds of FORTRAN. This permits AMTRAN to be used for efficient batch-processing. At the same time, AMTRAN's higher level instruction set results in a more compact code requiring less storage space than the machine language instructions generated by a FORTRAN compiler. (The reduced core required by the AMTRAN interpreter must be balanced against the storage requirements for the interpreter itself.)

3. Extensive error checks and diagnostic printouts are carried out during execution that can not occur with compiled object code.

4. Graphical input and output is available as an option for 1130 AMTRAN, using the low priced graphics terminals.

5. An interactive, numerical, analytical problem-solving system called AUTOMATH has been written in the AMTRAN language to facilitate the solution of numerical problems.

6. Various features have been incorporated into the language to facilitate compatibility with either interactive or batch processing. These include some symbolic capabilities, a text editor, and a built-in reference.

At the present time, AMTRAN is available to users at the Marshall Space Flight Center through the five IBM 1130 computers installed at MSFC. One of these machines has graphics terminals attached to it. For many small scientific and engineering problems, such as the evaluation of integrals or the solution of differential equations, 1130-AMTRAN can reduce problem-solving turnaround time to a few minutes.

An application for which AMTRAN was used at MSFC was in support of Dr. Lawrence Wood's doctoral thesis problem. Dr. Wood's problem consisted of computing the plasma current drawn by a charged space vehicle, with to-earth orbital workshop operations. Dr. Wood found that when trying to numerically integrate a curve with evenly spaced points, the results were not accurate to one decimal place (Fig. 13). This led to the development of the AUTOMATH subroutines in which the computer picks the interval sizes through accuracy tests that are too tedious for a human being to perform. The result generated by the AUTOMATH routines when applied to the curve shown in Figure 13 is depicted in Figure 14. The point density is visibly enhanced in the steep portion of the curve, and, in addition, there is an invisible guarantee of accuracy of better than 1 part in 10 million. This yields an integral with a guaranteed six decimal place accuracy.

The full set of operators in the AUTOMATH system are:

REPRESENT — Numerically represents mathematical formulas, selecting near optimum step sizes, assuming a cubic polynomial fit. Automatically detects singularities, cusps, discontinuties, and "hairpin" extrema.

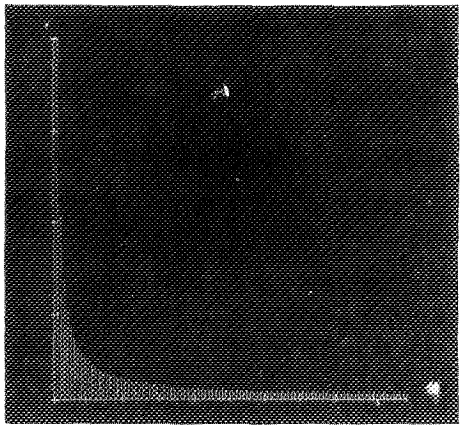


Figure 13. Curve with evenly spaced points.

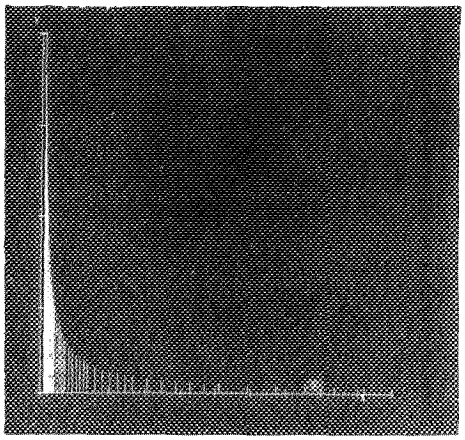


Figure 14. Result generated by AUTOMATH routines when applied to the curve of Figure 13.

$\int$ —— Numerically integrates mathematical formulas using variable-step-size error-controlled algorithms. Accommodates the general integral

$$\int_{g(x)}^{h(x)} f(x, x')\,dx'$$

including special cases such as

$$\int_{A}^{B} f(x')dx', \qquad \int_{x}^{B} f(x')\,dx', \quad \text{etc.}$$

DERIV —— Calculates the numerical derivative using a cubic fit (variable interval spacing).

SYMDIF — Symbolically derives the analytical derivative, given the analytical formula. The resulting analytical derivative is in the form of an executable code string, which may be evaluated to give numerical values for the derivative.

SOLVE — This operator solve sets of simultaneous algebraic or ordinary differential equations. The differential equation solver currently uses a variable-step-size, Runge-Kutta integration package developed at Aerospace Corporation that will handle any number of simultaneous first- or second-order differential equations. Error control is provided by a Simpson's rule check on the fourth-order Runge-Kutta integration formula. The algebraic equation solver currently uses the Crout reduction technique. It can also solve partial differential equations by the method of characteristics.

INVERT — The INVERT routine, when applied to a scalar, gives the reciprocal; when applied to a numerical representation of a monotonic function Y(X), gives the functional inverse X(Y); and when applied to a matrix A, gives the inverse matrix A^{-1}.

ZEROES — Locates all real zeroes within the range of definition of functions which have been generated by the REPRESENT operator. Gives warning if multiple roots are possible.

MINIMAX — Locates all relative extrema within the range of definition of functions which have been generated by the REPRESENT operator.

LET — The LET operator causes a numerical change of variables (for functions known only in tabular form).

STEP. FCT — STEP. FCT(T) is the unit step function defined by

$$u(t) = 0, \qquad t < 0$$

$$u(t) = 1, \qquad t \geq 0 \quad .$$

INTERPOLATE — Given two monotonic sets of numbers X and Y in one-to-one correspondence with each other, and a new set of x's (X1), the INTERPOLATE operator provides a new set of y's (Y1) corresponding to the X1's. A Newton third-order interpolation formula is used. The X1 array need not be the same size as the X and Y arrays and, for example, may consist of only a single number. The form is Y1 = INTERPOLATE X1, X, Y.

CUBIC — CUBIC accepts X and Y as inputs and generates four arrays A, B, C, and D of coefficients for the overlapping cubic

$$AX^3 + BX^2 + CX + D$$

fits for Y(X).

SCOPE — This nonmathematical automatic-formatting display operator may be followed by various modifiers and combinations of modifiers such as: POLAR, LOG. X, LOG. Y, VECTOR, GRID, HACHURE, MAGNIFY, etc. One or more curves may be plotted simultaneously with the scale factor automatically determined so that it is common to all the curves. In the absence of any such modifiers, the system displays the data in Cartesian form, selecting one of 25 different plotting formats, based upon the data, with printed scale factors and labeled axes.

SUM. OF. SERIES — The SUM. OF. SERIES operator is used to expedite the generation of functions by series expansions.

ERF(X) — The ERF(X) operator accepts an array expression as input and yields an array of error function values defined by

$$\frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\, dt$$

as the result.

LAPLACE — The LAPLACE operator accepts an array f(t) of exponential order as input and delivers the numerical representation of the Laplace transform, F(s), as output.

The AVERAGE, SIGMA, MOMENTS, REGRESSION, and CORRELATION routines are self-explanatory statistical operators. The LEAST, SQUARES operator provides the coefficients for a quadratic least-squares fit to numerical data Y(X). All the trigonometric and hyperbolic operators are also present.

Certain other operators based upon these fundamental algorithms are either available or are readily constructed.

Since 1130-AMTRAN is written in FORTRAN IV, a second way in which AMTRAN might be used is as a "conversational front end" for existing FORTRAN programs, to handle conversational interaction and graphical output. This has not been tried, but it appears promising.

A third way in which AMTRAN might be used is as a programming system for the construction of higher level, special-application languages. Since programming rates are high and since checkout is also swift, some engineering applications languages can be assembled in a short time.

APPROVAL

RESEARCH ACHIEVEMENTS REVIEW
VOLUME III REPORT NO. 9

The information in these reports has been reviewed for security classification. Review of any information concerning Department of Defense or Atomic Energy Commission programs has been made by the MSFC Security Classification Officer. These reports, in their entirety, have been determined to be unclassified.

These reports have also been reviewed and approved for technical accuracy.

DR. H. HOELZER
Director, Computation Laboratory

UNITS OF MEASURE

In a prepared statement presented on August 5, 1965, to the U. S. House of Representatives Science and Astronautics Committee (chaired by George P. Miller of California), the position of the National Aeronautics and Space Administration on Units of Measure was stated by Dr. Alfred J. Eggers, Deputy Associate Administrator, Office of Advanced Research and Technology:

"In January of this year NASA directed that the international system of units should be considered the preferred system of units, and should be employed by the research centers as the primary system in all reports and publications of a technical nature, except where such use would reduce the usefulness of the report to the primary recipients. During the conversion period the use of customary units in parentheses following the SI units is permissible, but the parenthetical usage of conventional units will be discontinued as soon as it is judged that the normal users of the reports would not be particularly inconvenienced by the exclusive use of SI units."

The International System of Units (SI Units) has been adopted by the U. S. National Bureau of Standards (see NBS Technical News Bulletin, Vol. 48, No. 4, April 1964).

The International System of Units is defined in NASA SP-7012, "The International System of Units, Physical Constants, and Conversion Factors," which is available from the U. S. Government Printing Office, Washington, D. C. 20402.

SI Units are used preferentially in this series of research reports in accordance with NASA policy and following the practice of the National Bureau of Standards.

CALENDAR OF REVIEWS

FIRST SERIES (VOLUME I)

REVIEW	DATE	RESEARCH AREA
1	2/25/65	RADIATION PHYSICS
2	2/25/65	THERMOPHYSICS
3	3/25/65	CRYOGENIC TECHNOLOGY
4*	3/25/65	CHEMICAL PROPULSION
5	4/29/65	ELECTRONICS
6	4/29/65	CONTROL SYSTEMS
7	5/27/65	MATERIALS
8	5/27/65	MANUFACTURING
9	6/24/65	GROUND TESTING
10	6/24/65	QUALITY ASSURANCE AND CHECKOUT
11	9/16/65	TERRESTRIAL AND SPACE ENVIRONMENT
12	9/16/65	AERODYNAMICS
13	9/30/65	INSTRUMENTATION
14	9/30/65	POWER SYSTEMS
15	10/28/65	GUIDANCE CONCEPTS
16	10/28/65	ASTRODYNAMICS
17	1/27/66	ADVANCED TRACKING SYSTEMS
18	1/27/66	COMMUNICATIONS SYSTEMS
19	1/6/66	STRUCTURES
20	1/6/66	MATHEMATICS AND COMPUTATION
21	2/24/66	ADVANCED PROPULSION
22	2/24/66	LUNAR AND METEOROID PHYSICS

SECOND SERIES (VOLUME II)

REVIEW	DATE	RESEARCH AREA
1	3/31/66	RADIATION PHYSICS
2	3/31/66	THERMOPHYSICS
3	5/26/66	ELECTRONICS
4	7/28/66	MATERIALS
5	9/29/66	QUALITY AND RELIABILITY ASSURANCE
6	1/26/67	CHEMICAL PROPULSION
7	3/30/67	CRYOGENIC TECHNOLOGY
8**	5/25/67	COMPUTATION
9	7/27/67	POWER SYSTEMS
10	9/28/67	TERRESTRIAL AND SPACE ENVIRONMENT
11	11/30/67	MANUFACTURING
12	1/25/68	INSTRUMENTATION RESEARCH FOR GROUND TESTING

THIRD SERIES (VOLUME III)

REVIEW	DATE	RESEARCH AREA
1	3/28/68	AIRBORNE INSTRUMENTATION AND DATA TRANSMISSION
2	5/22/68	ASTRODYNAMICS, GUIDANCE AND OPTIMIZATION
3	7/25/68	CONTROL SYSTEMS
4	9/26/68	AEROPHYSICS
5	11/21/68	COMMUNICATION AND TRACKING
6	1/30/69	THERMOPHYSICS
7	3/27/69	RADIATION PHYSICS
8	6/26/69	METEOROID PHYSICS
9	9/25/69	COMPUTATION RESEARCH
10	12/18/69	MATERIALS RESEARCH FOR SHUTTLE AND SPACE STATION
11	1/29/70	MICROELECTRONICS RESEARCH FOR SHUTTLE AND SPACE STATION
12	3/26/70	COMPUTATION RESEARCH

* Classified. Proceedings not published.

** Proceedings summarized only.

Correspondence concerning the Research Achievements Review Series should be addressed to:
Research Planning Office, S&E-R, Marshall Space Flight Center, Alabama 35812

MSFC—RSA, Aug